TRANSPARENT SECURITY SERVICES FOR AN INFRASTRUCTURE-AS-A-SERVICE CLOUD

SEBASTIAN BIEDERMANN

Dissertationsschrift in englischer Sprache zur Erlangung
des akademischen Grades Dr.-Ing. an der Technischen Universität Darmstadt

genehmigte Dissertation

Eingereicht von:
Dipl.-Inform. Sebastian Biedermann
geboren in Würzburg

Erstreferent: Prof. Dr. Stefan Katzenbeisser
Korreferent: Prof. Dr. Jakub Szefer
Tag der Einreichung: 19.12.2014
Tag der Prüfung: 13.02.2015

Security Engineering Group
Department of Computer Science
Technische Universität Darmstadt
Hochschulkennziffer D-17

Darmstadt, 2015

ABSTRACT

In this work we present novel security services in an "Infrastructure-as-a-Service" (IaaS) cloud computing scenario. The services can be offered by a cloud provider to customers according to a "Security-as-a-Service" (SECaaS) business model. The services operate transparently and benefit from techniques like Virtual Machine (VM) live cloning, VM introspection and VM storage forensics. They use unique strategies in order to analyze information or manipulate data which has been retrieved from a transparent hypervisor perspective and they run in a tamper-resistant environment separated from the systems of the cloud customers. We propose three different security services which detect attacks against the VMs of customers. They identify spreading malicious software, they detect phishing attacks executed with hitherto unknown spoofed web pages and they detect "Man-in-the-Browser" attacks. We propose two services which precisely analyze ongoing attacks as well as deceive attackers by dynamically extracting or redirecting them into honeypot-like environments. Furthermore, we propose two services which operate in a pro-active fashion by dynamically isolating sensitive data or by continuously implementing custom-made security settings.

ZUSAMMENFASSUNG

In dieser Arbeit präsentieren wir neuartige Sicherheitsdienste in einem "Infrastructure-as-a-Service" (IaaS) Cloud Computing Szenario. Ein Cloud-Provider kann Kunden die Dienste gemäß eines "Security-as-a-Service (SECaaS)" Business-Models anbieten. Die Dienste arbeiten transparent und profitieren von Techniken wie Virtual Machine (VM) Live Cloning, VM Introspection und VM Storage Forensics. Sie nutzen einzigartige Strategien, um Informationen zu analysieren oder Daten zu manipulieren, die aus einer transparenten Hypervisor-Perspektive gewonnen wurden und sie laufen in einer manipulationssicheren Umgebung, abgetrennt von den Systemen der Cloud-Kunden. Wir stellen drei verschiedene Sicherheitsdienste vor, die Angriffe auf die VMs von Kunden erkennen. Sie identifizieren sich ausbreitende bösartige Software, sie erkennen Phishing-Angriffe durchgeführt mit bisher unbekannten nachgeahmten Webseiten und sie erkennen "Man-in-the-Browser" Angriffe. Wir stellen zwei Dienste vor, die präzise laufende Angriffe analysieren und Angreifer täuschen, indem sie diese dynamisch in isolierte Honeypot-ähnliche Umgebungen extrahieren oder umleiten. Darüber hinaus stellen wir zwei Sicherheitsdienste vor, die auf eine pro-aktive Weise arbeiten, indem sie dynamisch sensible Daten isolieren oder kontinuierlich maßgefertigte Sicherheitseinstellungen umsetzen.

ACKNOWLEDGMENTS

First of all I would like to thank my supervisor Professor Stefan Katzenbeisser. His expertise, his continuous support and his helpful advice had a lot of impact on how I and my work developed in the last years. I would like to thank Professor Jakub Szefer, who gave me the opportunity to spend some time in the Computer Architecture and Security Laboratory (CAS Lab) at Yale University. There I enjoyed an inspiring research environment and participation in a lot of informative and very interesting projects. Furthermore I would like to thank my colleagues at the Security Engineering Group (SECENG) and at the Center for Advanced Security Research Darmstadt (CASED). With many of them, I had a lot of interesting discussions which often brought my work a step forward. Many of them became friends over the last years. Finally I would like to thank my parents, Sonja and Gottfried Biedermann. Without their support and their faith in me, nothing of this would have been possible. Last but not least, I would like to thank Alexandra Huck for continuously motivating me and consistently providing me with good spirits.

DANKSAGUNG

Zuerst möchte ich meinem Betreuer Professor Stefan Katzenbeisser danken. Seine Fachkenntnisse, seine kontinuierliche Unterstützung und sein hilfreicher Rat hatten großen Einfluss darauf, wie ich mich und wie sich meine Arbeit in den letzten Jahren entwickelt haben. Ich möchte Professor Jakub Szefer danken, der mir die Möglichkeit gab, einige Zeit am Computer Architecture and Security Laboratory (CAS Lab) der Yale Universität zu verbringen. Dort erlebte ich eine inspirierende Forschungsumgebung und konnte an vielen informativen und sehr interessanten Projekten teilnehmen. Weiterhin möchte ich meinen Kollegen in der Security Engineering Group (SECENG) und am Center for Advanced Security Research Darmstadt (CASED) danken. Mit vielen hatte ich zahlreiche interessante Diskussionen, welche meine Arbeit oft einen Schritt weiter gebracht haben. Viele wurden in den letzten Jahren zu Freunden. Zum Abschluß möchte ich meinen Eltern Sonja und Gottfried Biedermann danken. Ohne deren Unterstützung und Vertrauen in mich wäre nichts von all dem möglich gewesen. Nicht zuletzt möchte ich Alexandra Huck für die kontinuierliche Motivation und die durchweg entgegengebrache gute Laune danken.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PUBLICATIONS

Some of the ideas, of the text and of the figures have previously appeared in the following publications:

Chapter 2: Used Techniques and Related Work

- Sebastian Biedermann, Jakub Szefer,
  SystemWall: An Isolated Firewall using Hardware-based Memory Introspection, Proceedings of the 17th Information Security Conference (ISC), Oktober 2014

- Sebastian Biedermann, Martin Zittel, Stefan Katzenbeisser,
  Improving Security of Virtual Machines during Live Migrations, Proceedings of the 11th International Conference on Privacy, Security and Trust (PST), July 2013

Chapter 4: Transparent Attack Detection

- Sebastian Biedermann, Stefan Katzenbeisser,
  Detecting Computer Worms in the Cloud, Proceedings of the IFIP International Workshop on Open Problems in Network Security (iNetSec), June 2011

- Sebastian Biedermann, Tobias Ruppenthal, Stefan Katzenbeisser,
  Data-Centric Phishing Detection based on Transparent Virtualization Technologies, Proceedings of the 12th Conference on Privacy, Security and Trust (PST), July 2014

Chapter 5: Transparent Attack Analysis

- Sebastian Biedermann, Martin Mink, Stefan Katzenbeisser,
  Fast Dynamic Extracted Honeypots in Cloud Computing, Proceedings of the 4th Cloud Computing Security Workshop (CCSW), October 2012

- Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, Stefan Katzenbeisser, *From Patches to Honey-Patches: Lightweight Attacker-Misdirection, Deception, and Disinformation*, Proceedings of the 21st Conference on Computer and Communications Security (CCS), November 2014

Chapter 6: Transparent Attack Mitigation

- Sebastian Biedermann, Stefan Katzenbeisser,
  POSTER: Event-based Isolation of Critical Data in the Cloud, Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS), November 2013

- Sebastian Biedermann, Stefan Katzenbeisser, Jakub Szefer,
  Hot-Hardening: Getting More Out of Your Security Settings, Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC), December 2014

- Sebastian Biedermann, Stefan Katzenbeisser, Jakub Szefer,
  Leveraging Virtual Machine Introspection for Hot-Hardening of Arbitrary Cloud-User Applications, Proceedings of the 6th Usenix Workshop on Hot Topics in Cloud Computing (HotCloud), June 2014

1 INTRODUCTION

1.1 THE RISE OF CLOUD COMPUTING

In recent years, cloud computing became an IT-standard in the business world and the cloud is increasingly used in private life as well. While in private use cases people mostly use the cloud to backup and synchronize their data between different devices, companies outsource data as well as entire processes to cloud computing providers. This allows to save costs because the companies do not need to maintain an own hardware infrastructure nor do they need to recruit IT-specialists for administrative purposes.

However, the buzzword "cloud computing" often seems to be erroneously taken as a term which describes a technology. In particular, cloud computing is not a technology itself, rather it is a business model which has recently been facilitated by certain advances in various different technologies. These advances are in hardware, for example in the field of transmission technologies which allow higher and unlimited bandwidths for relatively low costs and in the field of processor technologies which became very efficient. Furthermore, these advances are also in software, especially in the field of virtualization technologies which allow to share physical computing resources between different software systems on a highly scalable, customizable and flexible way.

In general, the business value of cloud computing can be found in providing remote computing resources "as a Service" which means they are provided on demand and according to the principle of "pay-per-use". Established cloud business models are Infrastructure-as-a-Service (IaaS) giving customers privileged access to operating systems running in virtual machines, Platform-as-a-Service (PaaS) giving customers access to prepared and well-defined remote software appliances or Software-as-a-Service (SaaS) only giving customers access to specific applications that are hosted in the cloud. All these models provide high scalability and usability to customers as well as fine-grained billing methods and monitoring of costs.

In the foreseeable future, mobile clients in the form of lightweight notebooks, netbooks, readers, tablets and smartphones will continue to replace traditional computer systems and they will strongly depend on the scalable remote cloud computing resources to store and synchronize data or to use the cloud's computing power.

Unfortunately, in contrast to all the economic advantages which cloud computing introduced, security incidents occur on a daily basis and often on a large scale.

Foremost, the core components of cloud computing are standard network protocols and popular software components. Accordingly, the cloud is vulnerable against all attacks against which traditional computer networks are vulnerable as well. For example, programming flaws in the memory management can be found, sometimes even introduced by self-made software created by the customers in an IaaS cloud scenario [70]. The security impact of these vulnerabilities can be higher in the context of cloud computing because once a new vulnerability has been revealed, it can be usually exploited on a larger scale since the cloud is often a somewhat homogenous software environment. Especially, misconfigurations or frivolously deployed security settings by unaware customers who maintain privileges on cloud resources are a major reason for security incidents [78]. Often, cloud customers do not have the expertise to maintain complex distributed IT-systems especially considering security management and incident response.

However, besides the recurring software vulnerabilities, cloud computing also introduces new security issues which did not occur in traditional computer networks. For example, the world-wide distribution of data centers maintained by single providers and certain established (live) migration software techniques used by the cloud providers for optimization or incident response makes it difficult for customers to investigate the actual physical location of their outsourced processes which again makes it difficult to rely on nationally bound legislation [12]. Furthermore, the sharing of hardware resources with the help of software virtualization technologies enables the opportunity to mount certain side-channel attacks. For example, memory-based side-channel attacks have been proposed which can be used to retrieve sensitive information like encryption keys from operating systems of other customers [96]. Also, new privacy concerns occur, since when customers outsource data or their IT-processes to the cloud, they also sacrifice their control and they need to trust previously arranged agreements. In this context, the cloud provider's employees, (such as the server administrators) gain profound insights and unforeseen power [40]. Last but not least, based on the fact that everyone who is willing to pay can gain access to the cloud and its resources, attacks and misuse by malicious customers comes into play.

1.3 SECURITY OPPORTUNITIES INTRODUCED BY THE CLOUD

Fortunately, with the help of the underlying technologies used in cloud computing, existing security architectures can be improved and adapted to the new threats. Additionally, even novel security architectures can be developed and implemented.

Traditional security technologies can be adapted to the new properties of a cloud computing scenario and to the underlying software virtualization. For example, Trusted Computing (TC) techniques and protocols can be adapted and integrated into virtualization technologies ([30], [10]) or logging and monitoring techniques can be adapted by exploiting new sources of information in the cloud [92].

In addition, novel security architectures can be developed which benefit from the flexibility of the cloud's underlying software virtualization technologies, from the large scale of a cloud computing network or from an extensive new transparent monitoring perspective. For example, precise information about ongoing events can be retrieved through virtualization and this information can rapidly be analyzed and evaluated with the help of scalable cloud computing resources.

Furthermore, the network topology of the virtualized cloud network can be quickly adapted to new threats and the virtualized systems can be flexibly migrated [3], dynamically scaled, cloned [81] or even restored from (live) snapshots which is an ideal facility for security investigations or incident response. The novel security architectures can operate isolated and tamper-resistant in an own virtualized system separated from the customers' systems while being still able to monitor other virtualized systems located on the same physical hardware. These security architectures can benefit from a transparent perspective and even retrieve information of target running systems which traditional monitoring systems could not retrieve. The difficulty in building these novel cloud security architectures is mostly due to the interpretation of the data retrieved from a broad and transparent perspective.

1.4 THE INFRASTRUCTURE-AS-A-SERVICE BUSINESS MODEL

In this work, we design, implement and evaluate different novel security architectures for a cloud provider that offers cloud computing resources according to the Infrastructure-as-a-Service (IaaS) business model. The IaaS business model allows customers to get full on demand privileged access to operating systems installed in virtual machines (VMs) which are hosted on the hardware of a cloud provider. Customers can install new software, remove software and adapt all the available configuration settings. In the most cases, the customers can select VMs out of different performance classes. For

example, there are VM classes which are optimized for high performance computing with a lot of CPU power and a lot of memory, there are VM classes which are optimized for databases with large storage resources or there are VM classes which are optimized for scientific GPU computations.

A cloud provider offering IaaS is illustrated in Figure 1. Different customers get remote access to the cloud resources by accessing and using operating systems running in separated VMs hosted on the cloud provider's hardware. For the customers, costs incur depending on the time their VMs are hosted and depending on the selected VM class. Accounts and the resources of an IaaS cloud can be managed by a cloud administrator with the help of cloud management software, for example Openstack[1] or Eucalyptus[2].

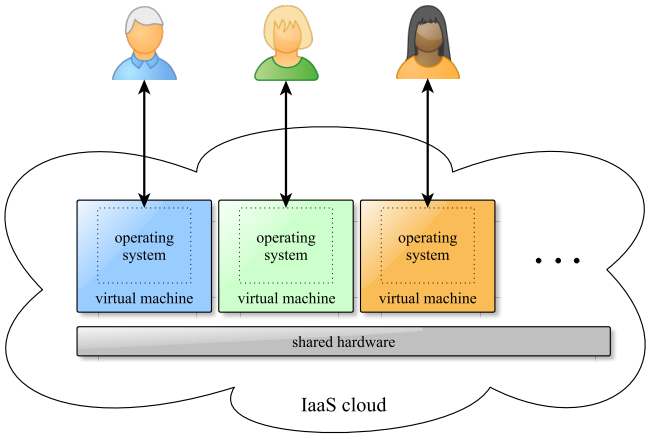

Figure 1: An Infrastructure-as-a-Service (IaaS) cloud provider offering access to remote operating systems to different customers.

Today, the leading cloud provider using an IaaS business model is Amazon with its Elastic Computing Cloud[3]. The IaaS business model is very successful, mostly because a cloud provider can offer cheap prices since optimized hardware can be effectively shared between different customers with the help of software virtualization. This way, the full hardware capacity can be continuously used. Furthermore, cloud providers can benefit from modern, centralized, large-scaled and optimized data centers.

1.5 Security-as-a-Service in Cloud Computing

In the following chapters, we present the design, the implementation and the evaluation of several different novel security architectures for an IaaS cloud which make use of the underlying technologies and which can be offered by a cloud provider to its customers. In

1 https://www.openstack.org/
2 https://www.eucalyptus.com/
3 https://aws.amazon.com/ec2/

particular, the proposed security architectures can be assigned to the "Security-as-a-Service" (SECaaS) business model which has lately become popular in the context of cloud computing.

There are various different definitions of SECaaS on the Internet. However, these definitions usually share certain core properties. Based on our investigations, security architectures which can be provided as a cloud service typically share three major features:

- The cloud security services are generic and support popular and widely distributed software components, platforms and different operating systems.
- The cloud security services are tamper-resistant and operate in a separated virtualized system isolated from the systems of the customers and isolated from the network used by them.
- The cloud security services only use transparent techniques to retrieve information about the systems of the customers. Customers do not need to install or configure any security software on their systems. The operations of the security services do not interrupt the work-flow of the systems of the customers nor they are detectable from their perspective.

Customers can select and enable specific security services on a pay-per-use basis, for example on the cloud provider's web page.

In this work, we present different novel security services (SECaaS), which can transparently detect, analyze and mitigate attacks against the systems of the IaaS cloud customers.

2 USED TECHNIQUES AND RELATED WORK

2.1 TAMPER-RESISTANCE AND TRANSPARENCY

As previously mentioned, we present novel security services for cloud computing which follow the paradigm of SECaaS (Section 1.5). These security services run isolated, tamper-resistant and retrieve information only transparently. In order to understand the concept of tamper-resistance and transparency in a computer security scenario, we briefly show a firewall architecture called "SystemWall" [11] that is realized with the help of a hardware extension. The SystemWall is no cloud security service nor it is based on virtualization technologies. However, it can be perfectly used to explain the terms tamper-resistance and transparency.

Many tamper-resistant and transparent security architectures use hardware expansions like a special trusted processor booting the micro kernel of the system [79] or particular co-processors only used for monitoring [39] or cryptography [2]. For example, Yashiro et al. [89] proposed to use a tamper-resistant chip which executes the sensitive operations in an access control scenario on a file system. In particular, tamper-resistant architectures are often used in embedded devices [69].

In the architecture for the SystemWall, we exploit Direct Memory Access (DMA) to create a firewall-like system which enables to transparently analyze the volatile memory of a system from an isolated perspective. DMA is part of the Peripheral Component Interconnect (PCI) specification which allows hardware devices to bypass the Central Processing Unit (CPU) and directly access the volatile system memory. This brings the advantage that a system's CPU can perform other useful tasks while DMA operations are in progress and it can accelerate certain hardware tasks. DMA has been in the focus of security researchers for some years, mostly because it easily allows to read the memory of a system through certain external interfaces while bypassing the security mechanisms of an operating system – and without any interruptions. In particular, DMA can be exploited by attackers to dump the volatile memory of unattended running systems through an external bus like for example ExpressCard, FireWire or Thunderbolt. Afterwards, the raw memory dump can be investigated using advanced forensic techniques in order to extract passwords, keys or other sensitive information. However, DMA can also be used to increase the security of a system and mitigate attacks. In particular, DMA can be used to transparently read the memory contents via an

external hardware device in a tamper-resistant perspective isolated from potential malicious software running on the target system. The contents can then be analyzed for malicious programs or for malicious established network connections – as the SystemWall does. The SystemWall can be deployed between a system and the Internet, and intercept all packets traveling from and to the target system. Placing the SystemWall between the Internet and the target system allows, for example, to delay network packets going to or from the target system while the target system's memory can be transparently analyzed in order to validate that the source of a network packet is legitimate and a non-malicious application. During this procedure, the SystemWall remains isolated and undetected. Figure 2 illustrates the architecture.

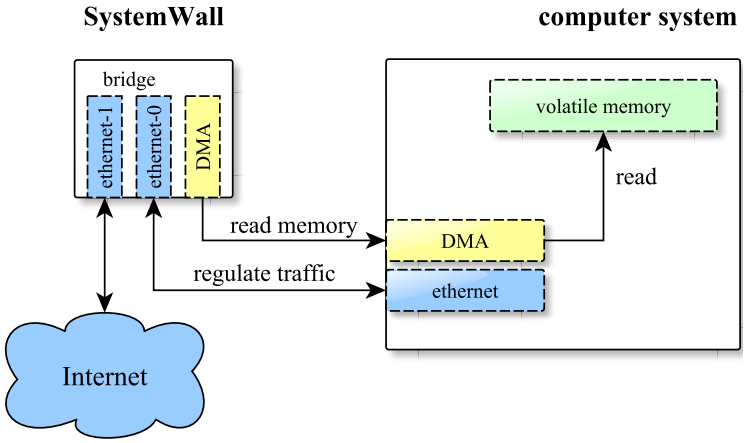

Figure 2: The SystemWall deployed as tamper-resistant and transparent device which analyzes a system's volatile memory and regulates its network traffic.

We assume that a transparent hardware-based memory analysis mechanism, such as through a dedicated PCIe expansion card, cannot be compromised by malicious software as it is separate from the target system. Furthermore, malicious software that manipulates the target operating system can be easily detected.

The SystemWall architecture is a nice example of how a tamper-resistant and transparent security architecture can look like. However, in addition to the advantages which tamper-resistance and transparency introduce, certain new difficulties arise. In particular, implementation is more difficult and sophisticated localization and tracing techniques are required to rapidly retrieve useful information from the raw data read in this perspective. Special strategies are needed which can make use out of the collected information. Furthermore, the information retrieved this way is often lagging behind the occurred events. If attacks should be prevented – and not only detected – other complementary protection mechanisms are required. For example, in the SystemWall scenario, new outgoing connections are de-

layed for a small period of time until the transparent analysis of the target system's volatile memory is finished.

In this section, we briefly introduced an isolated firewall device called SystemWall in order to explain the terms tamper-resistance and transparency in a computer security scenario. In the next chapters, we again focus on the development and evaluation of security services for an IaaS cloud computing scenario (Section 1.4). In this context, tamper-resistance and transparency is enabled with the help of software virtualization technologies instead of hardware extensions.

2.2 VIRTUALIZATION WITH THE XEN HYPERVISOR

In the scenario used in this work, we assume a cloud provider offering access to operating systems running in Virtual Machines (VMs) based on the Infrastructure-as-a-Service (IaaS) business model. In this scenario, multiple VMs of different customers run co-located on the same physical hardware and share hardware resources with the help of software virtualization technologies. Virtualization creates VMs, each VM emulates a stand-alone and isolated hardware container for an operating system. In general, virtualization is enabled with the help of a Virtual Machine Monitor (VMM), also called hypervisor. A hypervisor is a software component managing the sharing of computing resources and the isolation of multiple systems into VMs.

In this work, we use the Xen hypervisor ([8], [55]) which is a modern type-1 or "bare-metal" hypervisor. The Xen hypervisor runs directly on the hardware and creates VMs which run on a level above the hypervisor software. There are also other hypervisor architectures in which the hypervisor runs within an operating system (type-2 hypervisor) or there are hybrid hypervisor architectures[1,2]. In this context, the Linux Kernel-based Virtual Machine [77] (KVM[3]) is quite popular. We use the Xen hypervisor because it is open source software[4] and popular in academia as well as in industry. However, the proposed security services explained and evaluated in this work can also run with other hypervisor software.

The Xen hypervisor offers two ways in which VMs can be deployed: Para-virtualization requires certain modifications in the operating systems in order to support special hypercalls for communication between the hypervisor and the operating system as extensions to the systemcalls. This way, a higher performance can be achieved, because the hardware devices can be scheduled and no hardware platform needs to be emulated for a VM. However, we use the second type of virtualization offered by Xen which is called Hardware Virtual Ma-

1 VMware vSphere (http://www.vmware.com/)
2 VirtualBox (https://www.virtualbox.org/)
3 Kernel-based Virtual Machine (http://www.linux-kvm.org/)
4 The Xen project (http://www.xen.org/)

chine (HVM) or just "full-virtualization" [94]. In this context, full virtualization means that all attached hardware devices a VM uses are emulated. This brings the advantage that no modifications on the VM's operating system are required. Even a virtual BIOS is created for each VM. With this method, the operating system itself is unaware of being located and run in a VM. This allows to support proprietary operating systems, for example Windows. The Xen hypervisor relies on Qemu [9] to emulate a hardware setup for each VM that is deployed in HVM mode. Qemu is an open source machine emulator[5]. For the HVM mode, the used physical CPU needs to support an extended set of instructions called Virtual Machine Extensions (VMX) which basically allows to run the hypervisor code separated in a special root-operation mode while the VMs can run in non-root-operation mode. Using these extended instructions is called Hardware-assisted virtualization (Intel VT-x and AMD-V) and supported by CPUs of recent generations. A typical Xen hypervisor setup with several VMs (HVM) is illustrated in Figure 3.

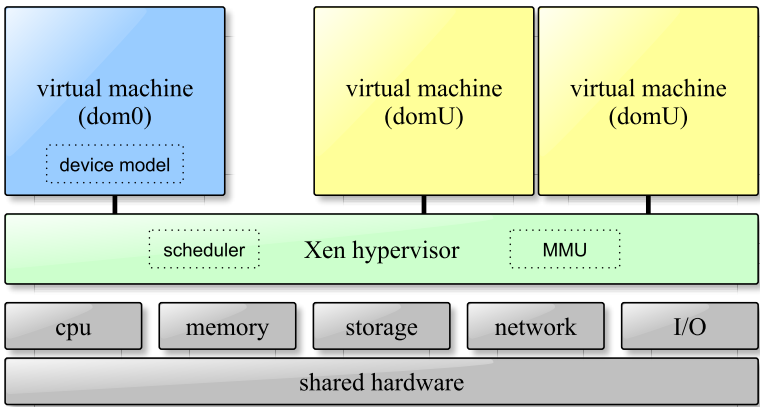

Figure 3: A Xen hypervisor setup with multiple virtual machines sharing a physical hardware platform.

In the Xen architecture, a VM called "dom0" (domain-0) is always present: It creates or destroys emulated devices and controls and maintains administrative privileges on the underlying Xen hypervisor. For example, the system running in dom0 is used by an administrator to start, scale or terminate guest VMs called "domU" (domain-User) which are assigned to the cloud customers in the IaaS business model (Section 1.4). The VMs share the CPU of the hardware with the help of Xen's scheduler, they share the available volatile memory with the help of Xen's memory management unit (MMU) and they share the storage, network devices and other available I/O devices. For example, the VMs can access the network through a virtualized bridge deployed and managed by Xen through virtualized network devices assigned to each VM.

5 http://www.qemu.org/

In order to enable the appreciated "pay-per-use" business model of cloud computing, the cloud provides a high level of scalability and flexibility. Especially in an IaaS cloud computing scenario, the concept of VMs allows to execute certain flexible operations which are not available in traditional computer networks that do not run virtualized systems. These operations are used for administrative tasks, but they can also be used for security purposes.

For example, the performance of VMs can be dynamically scaled during run-time, just by changing the underlying emulated hardware. This way, the size of the volatile memory can be increased or decreased while the operating system continuously runs. The VM's storage, the assigned virtual CPU (vCPU) units or any other emulated hardware device can be changed as well. In a Xen hypervisor setup (Section 2.2), the dynamic balancing of the volatile memory is quite popular and called "memory ballooning" [8]. Depending on executed operations and required hardware resources, the memory size can be dynamically changed and adapted.

Furthermore, a live snapshot of the current state of a VM and its operating system can easily be created during run-time. This way, the system can be restored from the snapshot at a later point in time. This operation is especially interesting for backup purposes, but also for security purposes, for example for incidence response [41].

Another interesting operation provided by virtualization technologies is live migration of VMs. Clark et al. [18] proposed the first live migration algorithm for the Xen hypervisor in 2005. The proposed strategy allows to migrate a running VM to a new physical hardware platform with only very minimal and negligible downtime. No operations as well as no established network connections are interrupted. Live migration is a useful tool for performance optimization tasks within a local area network and for incident response. For example, if hardware failures occur or if important upgrades need to be installed which require a reboot of the hypervisor, the VMs can be flexibly migrated to other hardware.

In a live migration procedure, the target VM's storage is not relocated, only the VM container defining its emulated hardware including the VM's volatile memory is transferred. In a cloud computing setup, the storages of VMs are usually located on a centralized network file system (NFS) which can be remotely accessed by the hypervisor nodes. Figure 4 shows a Xen hypervisor setup enabling live migration of VMs. The figure shows a customer VM with ID-2 being live migrated from the physical computing node 1 to the physical computing node N during run-time while the storage of the VM remains on a storage server.

A live migration procedure only causes a negligible downtime. Running processes, currently executed tasks as well as established network connections of the live migrated VM are not interrupted. Figure 5 illustrates the six sequential steps performed by a Xen hypervisor in a live migration procedure.

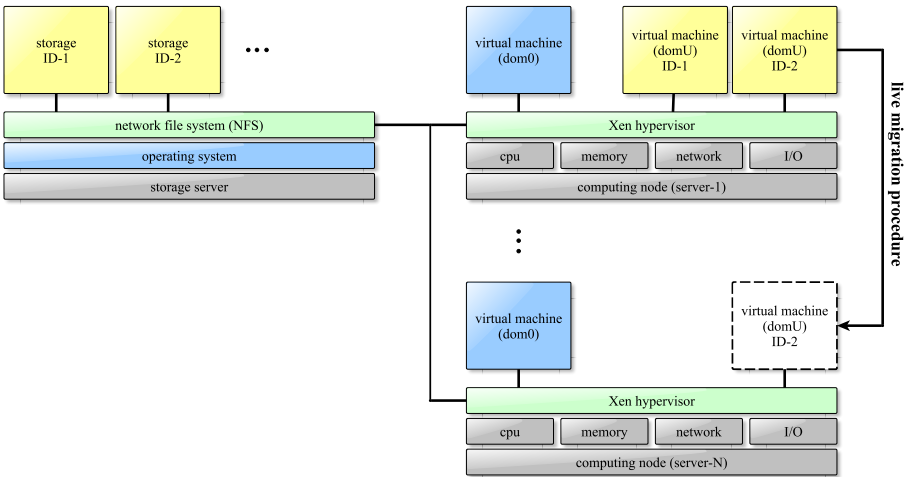

Figure 4: A Xen hypervisor setup which enables live migration of virtual machines between different physical nodes.

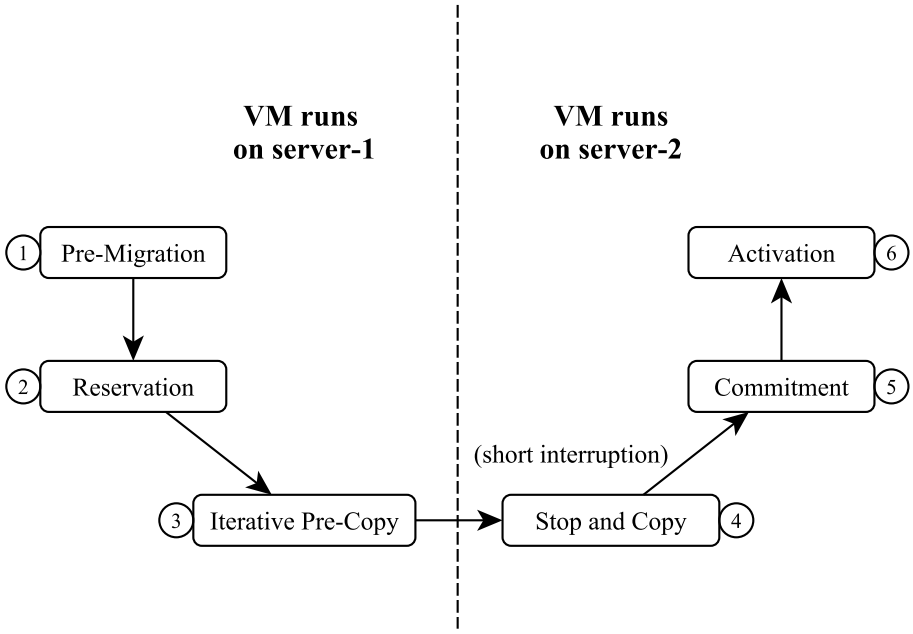

Figure 5: Subsequent steps of a Xen VM live migration procedure.

In the first step (pre-migration), a new remote hardware node is selected and the required resources are allocated. In the second step (reservation), the new allocated resources are used to initialize a new virtual container providing the required emulated hardware which is used to deploy the VM. If this step was successful, the volatile

memory of the target VM is copied from the source server to the new server in the third step (iterative pre-copy step). The VM which is about to be live migrated continuously runs and accordingly its volatile memory gets modified depending on current tasks. After the first copy process, the Xen hypervisor checks for modifications in the VM's volatile memory on the source server caused by the running VM and potential operations executed during the first copy process. Identified so called "dirty" memory pages are again transferred to the new server and the hypervisor again checks for new modifications. This procedure is iteratively repeated until only a few memory pages are not synchronous. In step four (stop and copy), the VM on the source server is stopped and the remaining dirty memory pages are transferred. This interruption is required to have a fully synchronous memory copy on the new physical server. The VM's operation is only interrupted for a very small period of time in the range of milliseconds and currently running processes or established network connections are not affected. A logged-in remote user does not recognize the interruption nor the live migration procedure in general. In step five (commitment), the VM on the source server is terminated and the hardware resources are freed. Finally, in step six (activation), the VM on the new server is activated. During the live migration procedure, the storage of the VM remains on the network file system (NFS) and is not migrated. Network management is organized by the Xen hypervisor and the migrated VM maintains the same IP address.

After integrating the live migration procedure into the Xen hypervisor, several work has been published which analyzed the performance [45] or which improved the proposed algorithm, for example by benefiting from data de-duplication techniques [95] or by using a slowdown scheduling algorithm that decreases the CPU resources of the migrating VM and accordingly reduces the dirtying page rate. This approach lead to a faster live migration procedure [53]. Other work analyzed the energy efficiency and the power consumption during a live migration procedure [37] or optimized the energy consumption by determining the best source and target physical server [87]. Furthermore, not only live migration in local area networks, but also wide-area live migration was the target of research [14]. This topic is especially interesting for cloud providers which operate in distributed data centers in different countries. Another interesting application of live migration was proposed by Wang et al. [86] who reduced the complexity of network management tasks by introducing moving routers.

The first security-related issue based on empirical exploitation of the live migration procedure was presented in 2008. Oberheide et al. [59] analyzed Man-in-the-Middle (MitM) attacks by manipulating the VM's volatile memory during a live migration procedure with the help of a malicious modified router. Furthermore, software bugs

which lead to security problems have been discovered in the implementation of the Xen live migration algorithm and exploits can be found on the Internet (e.g. CVE-2013-4356).

On the other hand, architectures based on live migration procedures improving security have also been proposed. Ando et al. [3] proposed a reliable web server protection system which uses live migration to adapt the internal network topology in case of Denial of Service (DoS) attacks. In general, fast changes on the internal network topology based on live migration is a promising approach targeting security-related issues. For example, if a VM is the target of a DoS attack, it can be migrated to physical hardware having better performance. Zhang et al. [93] proposed a secured live migration architecture with small performance degradation for VMM-enforced protection networks. Gondree et al. [31] proposed a data geo-location protocol which is a combination of network latency measurements and provable data possession. Their solution helps to bind outsourced data to specific geographical regions. Finally, also work of us can be mentioned here, which deals with the early detection of a live migration procedure in order to enforce location depending security policies in time [12].

Since the introduction of live migration, also live cloning approaches have been proposed for the Xen hypervisor inspired by the live migration algorithm [81]. While a live migration procedure relocates a running VM to new hardware, live cloning creates a consistent replica of a running VM and its operating system on other hardware or on the same hardware. Basically, a live cloning procedure is similar to a live migration procedure, except that the source VM is not destroyed and the live cloning procedure needs to take particular care to ensure correct resource management, especially if the cloned VM operates on the same physical hardware. In particular, assigning a valid networking configuration and communication issues in general are difficult to handle. In this context, the clone should not use the same IP address. Live cloning can be particularly interesting for various security purposes. For example, as proposed by Lagar-Cavilla et al. [47], live cloning can be used to fork a VM and its tasks or it can be used to test and verify a new software configuration on a cloned VM before it is finally deployed on the original VM [97].

The cloud provides high scalability based on procedures like VM live migration or VM live cloning. These operations are enabled by the underlying software virtualization technologies. These operations lead to rethink and to revisit some ideas and implementations of certain security architectures, for example in the field of incident response or attack analysis. They can be used to improve existing security architectures and they can be used to develop new security architectures for the cloud.

In this work, we often use the technique of Virtual Machine Introspection (VMI) which is facilitated by software virtualization. VMI allows to transparently monitor the operation of a target VM by directly reading the VM's volatile memory pages during the VM's operating system runs. VMI operations are executed from an isolated and privileged position which is the administrative VM (dom0) in the Xen setup hosted on the same physical hardware together with multiple customer VMs. VMI is especially interesting for security purposes like for example intrusion detection which can be implemented transparently from a tamper-resistant perspective and unbeknownst to the target VM being monitored.

In general, with the help of VMI, security architectures can be moved to a separated and isolated VM while they can still operate on the memory of co-resident VMs without being detected. In our IaaS cloud computing scenario, the privileged Xen VM (dom0) can be used to transparently monitor the volatile memory of VMs of the customers. Figure 6 illustrates the perspective of VMI in the Xen hypervisor setup (Section 2.2).

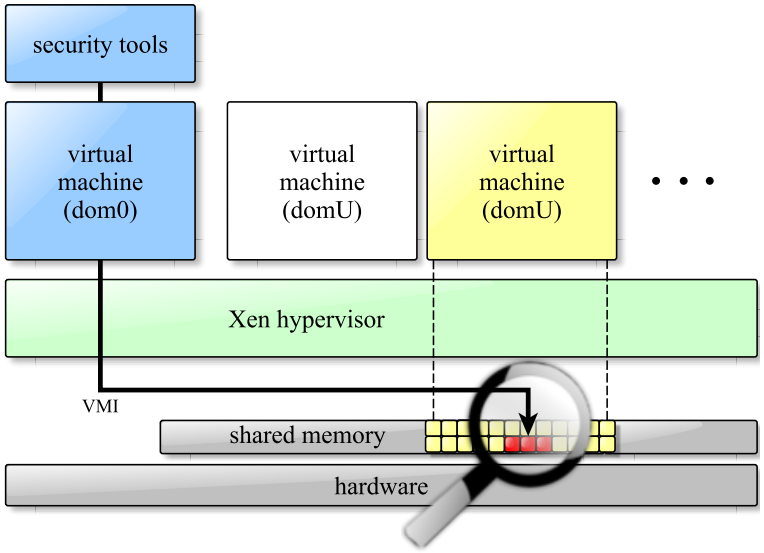

Figure 6: Virtual Machine Introspection (VMI) used to transparently investigate the volatile memory of a running Xen guest VM (domU) from the perspective of the privileged VM (dom0).

The difficulty of VMI is due to the localization of the memory used by a VM and the extraction of useful information from this continuously changing volatile memory region. For example, processes operating in a customer VM are constantly allocating or de-allocating volatile memory pages depending on their tasks or depending on the current work load.

Figure 7 illustrates the virtual memory regions which use different physical memory addresses that need to be investigated in order to extract useful information from the virtual address space assigned to a target process running in a customer VM. In general, the volatile memory registers can be read by using physical memory addresses, virtual addresses managed by the Memory Management Unit (MMU) of the Xen hypervisor, or by virtual addresses managed by the Memory Management Unit of the installed operating system.

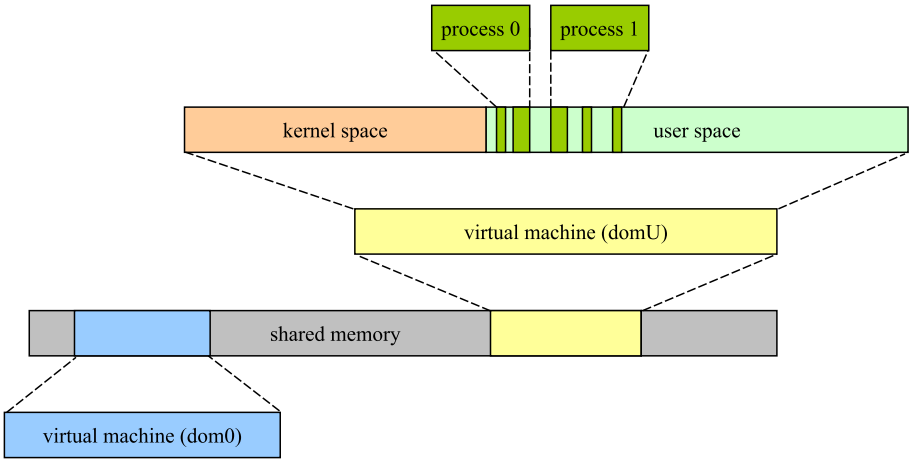

Figure 7: Different memory regions on a VM (domU) which can be transparently investigated with the help of VMI.

Xen provides a function called *xc_map_foreign_range()* which can be used to map memory of a VM to another VM. However, Xen supports only to access the memory using low-level Machine Frame Numbers (MFN) which are cumbersome to handle. Payne et al. [63] introduced a VMI library[6] for the Xen hypervisor which we often use in this work. The VMI library provides a higher level of abstraction by translating addresses. It can convert Physical Frame Numbers (PFN) to MFNs with the help of Xen's internal look-up table or it can convert virtual addresses to the corresponding MFNs with the help of the target operating system's internal page table. The library helps to find the correct address ranges of the volatile memory regions of a target VM which then can be precisely investigated with other methods. Furthermore, the library provides a nice Python interface. In order to extract useful information from the raw data stored in the volatile memory, the so called "semantic gap" needs to be bridged. This means certain previous knowledge about the target VM's operating system is required, for example the location of the VM's page table needs to be known which can then be walked in order to retrieve further dynamically stored data within a customer VM's volatile memory [22].

Since VMI only enables to transparently access the raw volatile memory regions of a co-resident VM, VMI techniques need to be com-

6 https://code.google.com/p/vmitools

bined with advanced memory forensics techniques in order to extract useful information [58]. Traditional memory forensic techniques are basically used to analyze a dump of an operating system's volatile memory. However, in a hypervisor setup, we can tunnel advanced memory forensic operations through VMI and execute them on a virtualized system's volatile memory during run-time. In this work, we use the memory forensic framework Volatility[7] which can be combined with VMI for advanced memory analysis on running VMs. Volatility provides OS-dependent modules that analyze raw memory data and extract information. For example, if a target VM runs Windows and the static virtual address of the Windows kernel symbol *PsInitialSystemProcess* and certain offsets are known, for example defined in a profile, the combination of VMI and memory forensics can be used to transparently extract a list of currently running processes on the system. As another example, the tree structure of the Windows virtual address descriptors [21] can be examined and the memory pages assigned to a specific process can be located and investigated [52].

In general, some work has been done in the field of security architectures which use VMI techniques. Garfinkel et al. [29] presented an IDS architecture maintaining the visibility of a host-based IDS, but moving the IDS software to the outside of the monitored system in order to increase attack resistance. Ibrahim et al. [38] proposed CloudSec, a fine-grained monitoring architecture which transparently monitors the dynamic kernel data structures of operating systems running in VMs. Azmandian et al. [6] built a transparent IDS which uses data mining techniques in order to to extract and analyze useful data from the raw memory regions. Crawford et al. [19] presented a transparent security architecture which aims at identifying complex potentially malicious actions of insiders by using VMI and an attacker taxonomy. Fraser et al. [27] implemented an immunity architecture that can detect kernel-modifying attacks on VMs with the help of integrity measurements and autonomously repair the affected memory parts from a transparent and tamper-resistant perspective during run-time. More et al. [57] focused on the transparent detection and collection of malware samples. Recent research in the field of VMI focuses not only on the transparent reading of a target VM's volatile memory, but also on writing to specific memory pages during run-time at certain specific points in time. This way, even new procedures can be dynamically implanted and the execution can be enforced [33]. However, active VMI needs to have very accurate and precise information about the target system's internals and operations in order to avoid inconsistent states and crashes.

7 https://code.google.com/p/volatility/

The hypervisor setup enables to transparently investigate the volatile memory of a VM during its run-time from an isolated and tamper-resistant perspective. Additionally, this perspective also allows the examination of a VM's virtualized storage – again transparently and during run-time. Similar to the transparent investigation of the volatile memory, certain previous information about the type and the structure of the target virtual storage is required in order to extract useful information from the raw data. For example, the size of the installed boot sector needs to be known as well as the type of the used file system (for example NTFS, FAT32 or EXT3), the block size and the sector size. Once this information is available (or can be retrieved by testing), the file system of a target VM can be transparently examined and data can be located, read and modified. Figure 8 illustrates a VM's virtualized storage device stored on the physical storage device on a server managed by the Xen hypervisor.

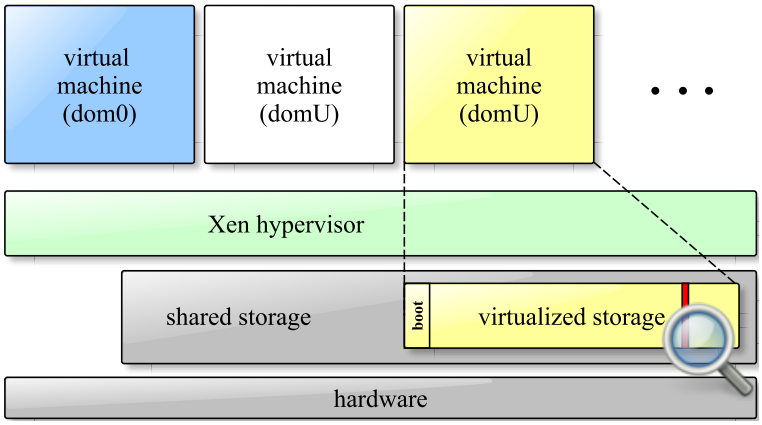

Figure 8: A virtual storage device assigned to a VM (domU) and hosted on a physical storage device managed by the Xen hypervisor.

From the perspective of the administrative VM (dom0), the storage of a customer VM is a raw data file from which data needs to be precisely located before it can be read – no matter whether the target VM to which this virtual storage is assigned is running or not. In this work, we use the Sleuthkit[8] storage forensic tools to examine a target VMs storage. These tools help to investigate the file system and to locate meta information of specific target files which again help to locate the actual data of these files.

For example, the inode information (index node) which represents a file system object on a journaling file system, (e.g. EXT3) can be investigated and additional information like required permissions to access this file or the last time the file was accessed can be retrieved. Furthermore, the inode information can be used to find the corre-

8 http://www.sleuthkit.org/

sponding blocks of the actual data on the raw storage. The block information can in turn be used to compute the target file's corresponding sectors on which the target data is finally stored. Using this information, the actual position of the data of a target file on the raw storage can be calculated.

For example, for a known block number n_{block} which has been retrieved from the file system's meta information, for a known boot sector size s_{boot}, for a known block size s_{block} and for a known sector size s_{sec}, the sector n which finally stores the data of the target file can bey calculated with $n = \frac{(n_{block} \cdot s_{block} + s_{boot} \cdot s_{sec})}{s_{sec}}$. Finally, the data stored on the sector can be transparently read or even replaced with new data just by directly overwriting the sectors out of the perspective of the administrative VM (dom0).

If data is transparently examined on a virtual storage during runtime of the corresponding VM's operating system, some special characteristics need to be considered. An operating system is usually not directly writing data to a hard drive because of performance issues. Rather it buffers the data of the executed read or write operations in a page cache in its volatile memory. This way, multiple write operations to a hard drive are combined to single bigger write operations or data which is frequently read is kept in the volatile memory and can be accessed faster. However, this leads to the effect that data which is written by a VM's operating system to its storage can only be read delayed from the perspective of the administrative VM (dom0).

There are certain parameters in an operating system which define how long data is kept in the page cache or how much data is buffered before it is finally written to the hard drive. For example, in Linux, parameters like the *dirty_expire_centisecs* or the *dirty_writeback_centisecs* which can be set in */proc/sys/vm/* define caching properties of the file system. Fortunately, these parameters are set to known values by default and they are usually not modified. Additionally, they are stored in the operating system's volatile memory and this way, they can be transparently modified with the help of VMI (Section 2.4). In particular, delayed operations on the virtual storage need to be considered when using transparent storage forensic techniques in our security services. Furthermore, if we transparently overwrite the sectors of a file in order to replace the stored data with new data, we need to take into account which type of file system is used by the VM's operating system. The most popular file systems do not perform integrity measurements on the data, however, there are file systems which compute checksums and can this way reveal any modifications (for example Oracle's ZFS file system).

3 THESIS ORGANIZATION AND CONTRIBUTIONS

So far, we introduced cloud computing (Section 1.1) and our scenario which is an Infrastructure-as-a-Service (IaaS) cloud provider that offers virtual machines (VMs) to customers on demand and according to the principle of "pay-per-use" (Section 1.4). We briefly outlined certain security issues of cloud computing (Section 1.2) and new opportunities introduced be the cloud and its underlying technologies which can be used for security purposes (Section 1.3).

Cloud providers offering the IaaS business model are rapidly growing and security architectures which are especially tailored towards this scenario are required. In particular, there is a high demand for flexible Security-as-a-Service architectures which are basically characterized by their transparent and generic mode of operation (Section 1.5). Besides the wish to secure and to protect the systems of the customers on a better way, these services are a great opportunity for business as well.

In order to design and implement transparent security services, different problems need to be solved. Techniques are required which timely and precisely locate information in the data transparently read from the raw memory and the raw storage or captured on emulated devices during run-time. Strategies are required which interpret and make use out of this information, especially regarding security. Complementary mechanisms are required which actively intervene in order to prevent attacks or implement protection mechanisms.

In this work, we present several novel security architectures for the IaaS cloud computing scenario which benefit from techniques like VM live cloning (Section 2.3), VM introspection (VMI) (Section 2.4) and VM live storage forensics (Section 2.5). We designed, implemented and evaluated the security services in a Xen hypervisor setup (Section 2.2). However, the proposed strategies and techniques can also be used in other hypervisor setups. Our security services operate transparently and run in a tamper-resistant and isolated environment separated from the systems of cloud customers (Section 2.1). Our services use novel, unique and well-defined strategies to analyze information retrieved in a transparent perspective. The services can be offered by a cloud provider to its customers to protect their systems. In order to provide a broad security infrastructure for an IaaS cloud, our proposed security services cover three different major areas in the field of computer security:

- **Transparent Attack Detection:** In Chapter 4, we propose three different security services which detect attacks against the VMs

of the cloud customers in the early stages. This brings the advantage, that a cloud provider can intervene in order to prevent damage. Each of the security services solely retrieves and analyzes specific information transparently with the help of VMI (Section 2.4). Furthermore, each of the three services uses a unique strategy for attack detection. The first service detects unknown spreading malicious software which was has not been examined or analyzed before. The second service detects phishing attacks targeting the customers based on hitherto unknown spoofed web pages and the third service detects Man-in-the-Browser (MitB) attacks.

- **Transparent Attack Analysis:** In Chapter 5, we propose two different security services which precisely analyze ongoing attacks as well as deceive attackers by dynamically extracting and redirecting them into isolated honeypot-like environments. These services use a combination of transparent techniques, basically VM live cloning (Section 2.3), VMI (Section 2.4) and VM live storage forensics (Section 2.5). When using the services, customers learns from attack techniques and can reveal the intentions and strategies of attackers while their systems remain unaffected and secured.

- **Transparent Attack Mitigation:** In Chapter 6, we propose two different security services operating in a pro-active fashion. The services use active VMI (Section 2.4), which means they do not only transparently read and analyze the volatile memory regions of a customer systems, but they also write to specific locations during run-time in order to insert new data. The first service mitigates and prevents attacks by isolating, securely storing and only temporarily injecting sensitive data into customer VMs – once it is required and only as long as it is required. The second service periodically and autonomously deploys optimized and custom-made security settings for network applications running on customer VMs.

Figure 9 illustrates the three different security areas covered by our seven different security services in the IaaS cloud computing scenario. Besides other services, the cloud provider can flexibly offer our security services to its customers based on a pay-per-use business model.

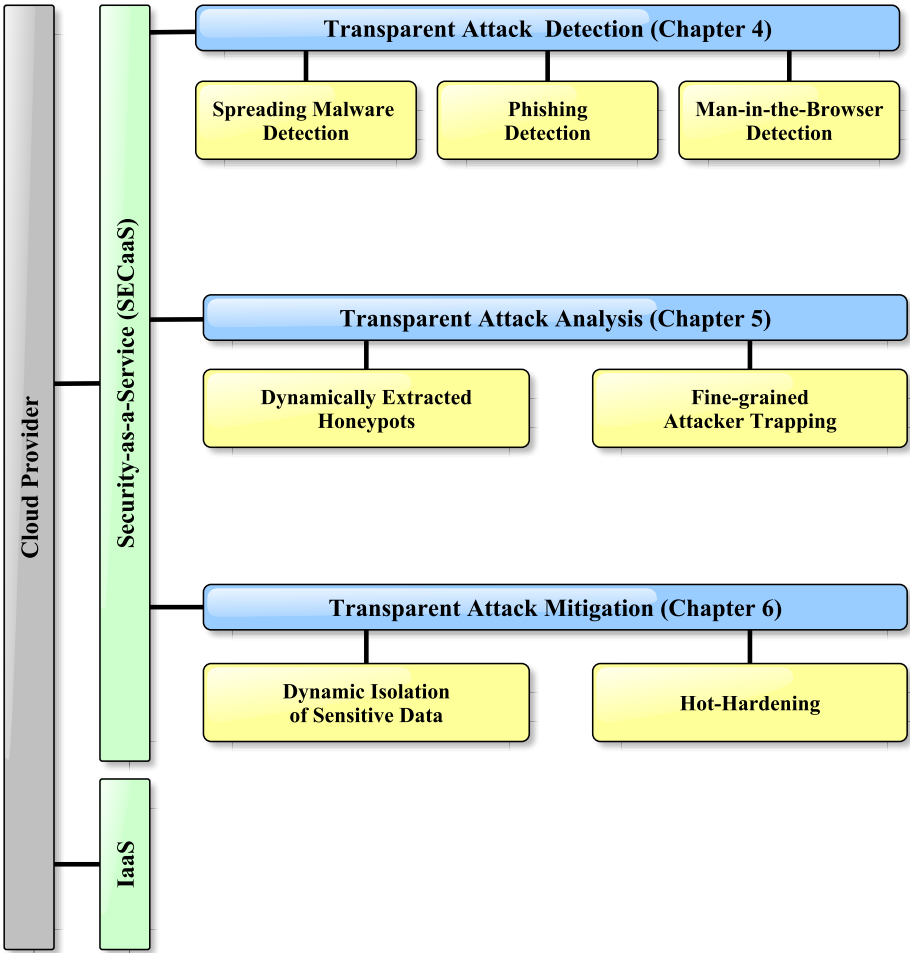

Figure 9: Illustration of the three different security areas covered by our seven different security services for an IaaS cloud scenario.

4

TRANSPARENT ATTACK DETECTION

In this chapter, we introduce three different security services which detect attacks against systems of cloud customers. The first security service detects malicious software (malware) by transparently performing integrity measurements on the customers' software setup and by detecting spreading behavior of unknown processes among different systems (Section 4.1). The second security service detects and mitigates phishing attacks which are executed with the help of hitherto unknown spoofed web pages (Section 4.2). The third service detects attacks which use a Man-in-the-Middle strategy, in particular it can detect Man-in-the-Browser attacks (Section 4.3).

4.1 DETECTION OF UNKNOWN MALICIOUS SOFTWARE

In general, malware is an umbrella term which covers a broad number of different malicious software components, for example computer worms, viruses, trojan horses, backdoors or rootkits. All these kinds of malware use different malicious strategies as well as they have different objectives. Malware is used to gain access to or manipulate a target operating system, to record, collect and forward sensitive data (for example passwords or banking information) or to abuse the infiltrated system for other attacks like Distributed Denial-of-Service attacks (DDoS). Often, malware is created for espionage, for financial gain, for extortion or for sabotage.

For example, in November 2008, the "Conficker" computer worm infected up to 15 million Windows systems by remotely exploiting a software vulnerability in the Network Basic Input Output System (NetBIOS). The worm injected a randomly named Dynamic Link Library (DLL) into an authentic Windows process (svchost.exe) [48]. Version A of Conficker focused on the infection of operating systems connected within the same internal network. This resulted in fast propagation and high network traffic within local networks of organizations. Version B additionally used brute force attacks to guess passwords of other network services and version C of Conficker even used a P2P protocol in order to manage and optimize its own propagation [32]. In the year 2010, a new computer worm named "Stuxnet" ushered in a new era of autonomously spreading malware. This computer worm aimed to manipulate industrial control systems which are used for Supervisory Control and Data Acquisition (SCADA). Stuxnet was able to attack a range of different Windows operating systems (version 2000 up to 7) by using different known and unknown

exploiting techniques [24]. The occurrence of Stuxnet and its revised successors Duqu (2011) and Flame (2012) showed that the danger caused by spreading malware is far away from being over.

Constantly, new autonomous computer worms are created and propagate, although sophisticated detection and mitigation techniques are used. Even detection software like anti-virus software installed on the affected operating systems can be manipulated by the malware or can be misconfigurated by unaware users. Furthermore, detection software is often based on a large up-to-date database of signatures of captured and previously analyzed malware. However, these databases grew increasingly faster and maintenance of an up-to-date database becomes more and more challenging. Mitigation of unknown autonomously spreading malware is still an open problem which we address in this first security service for the IaaS cloud.

In traditional computer networks, each operating system is installed on a single hardware component. On the contrary, the cloud computing network is virtualized and multiple operating systems run in parallel on a single physical hardware platform, managed by a hypervisor (Section 2.2). In the cloud, malware can still infect the customers' operating systems and spread for example by exploiting software vulnerabilities in the network applications. Computer worms which autonomously infect systems within a cloud computing network can even propagate faster and on a larger scale because the cloud environment is often quite homogenous with a lot of similar software installations. However, by benefiting from the underlying virtualized infrastructure of the IaaS cloud, information about running systems can be retrieved faster, more easily and more extensive.

In the most incidences, malicious software (malware) is injected into operating systems either by exploiting software vulnerabilities, misconfigurations or social engineering strategies such as emails having spoofed senders. Since the customers of the IaaS cloud gain full administrative privileges over the operating systems running in VMs, they are also responsible for security updates and correct software configuration. However, cloud customers do not always have the expertise in the field of security nor is every customer frequently deploying urgent security updates. This can finally lead to compromised systems and infections with malware within the IaaS cloud network.

In this section, we introduce a security service which detects hitherto unknown propagating malware and this way protects the systems of the cloud customers. Our proposed security service operates transparently and can be easily and beneficially implemented in existing cloud architectures.

4.1.1 *Transparent Integrity Measurements*

A popular approach to detect unknown malware are integrity measurements performed on a software setup. In order to verify the integrity of a running application of a cloud customer, we verify the executable code of the corresponding process stored in the VM's volatile memory during run-time. In most operating systems, each process runs in its own virtual address space which is mapped to physical memory addresses by a page table maintained by the operating system. In a hypervisor setup, a second layer of memory addresses comes into play which is managed by the underlying hypervisor software and used to create a virtual memory container for each VM. In order to measure integrity, we first need to locate the memory pages storing the executable code of the target process. In Linux, the executable code of a process is stored in the Executable and Linkable Format (ELF) in the volatile memory in the text region. Figure 10 illustrates the different memory regions which need to be investigated in a hypervisor setup.

First, the physical memory region assigned to a customer's VM needs to be retrieved. Subsequently, specific statically stored information from the operating system's kernel is read which finally allows to locate the different memory pages assigned to a target running process in the userspace. Finally, the memory pages which store the executable code can be located. Other memory pages assigned to the process are for example the stack, the heap or mappings. In this section, we are only interested in the executable code. For localization, previous information about the static internals of the used kernel defined in a profile is required (Section 2.4). The currently allocated and used memory pages of the target process are transparently read from the co-located administrative VM (dom0) and the integrity of the executable code can be verified.

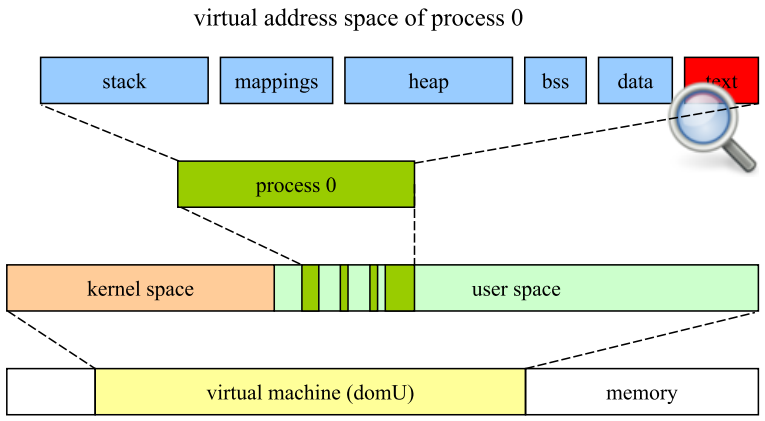

Figure 10: Different memory regions investigated in order to read the executable code (ELF) of a target process from a transparent hypervisor perspective.

After localization, we compute a cryptographic hash function over the data and compare the result with a hash of the application stored in a white-list in a database in the administrative VM (dom0). This way, the integrity of the process can be transparently verified during run-time and any modifications on the executable code in the memory, for example executed by malware as well as any unknown and not white-listed running process are revealed.

The integrity measurement of a cloud customer's process can be used as the last element in a chain of subsequent verifications of a hypervisor setup. In the concept of Trusted Computing (TC) introduced by the Trusted Computing Group[1], a chain of trust is used to verify the integrity of an entire software platform. The chain uses a trust anchor as "root of trust". This anchor is a hardware device called Trusted Platform Module (TPM). The chain of trust is subsequently created by performing integrity measurements over the concatenation of a previous hash and a new hash over a piece of code.

However, in this work, we do not use a chain of trust or a TPM. Instead, we assume a trustworthy hypervisor setup and a trustworthy co-located administrative VM (dom0) which can not be accessed from the network used by the cloud customers. In this proposed security service, we use the transparently retrieved integrity measurements of running processes to detect unknown spreading malware among the operating systems of different customers.

4.1.2 *Identifying the Propagation of Malware*

The security service transparently monitors properties of running processes in different customers' operating systems with the help of VMI (Section 2.4). The service can not be manipulated by malware which successfully compromised an operating system because the service operates isolated on the administrative VM (Figure 11).

In particular, software virtualization offers an elegant way to identify spreading malware without the need to have previously generated signatures. Our service randomly monitors VMs of different customers and interprets the status of the entire cloud network by using a centralized superior perspective. Unknown spreading malware is detected only by identifying the spreading behavior of an effect caused by the malware on a constantly increasing number of different operating systems. Once detected, a signature is generated and used in traditional network traffic filtering software.

We first define an anomaly which is tracked by our security service within the cloud network: *An anomaly is an unknown running process* P *of which the integrity could not be verified by comparing the hash over* P*'s executable code stored in the volatile memory with white-listed hashes of known benign processes stored in a database* (Subsection 4.1.1).

1 http://www.trustedcomputinggroup.org/

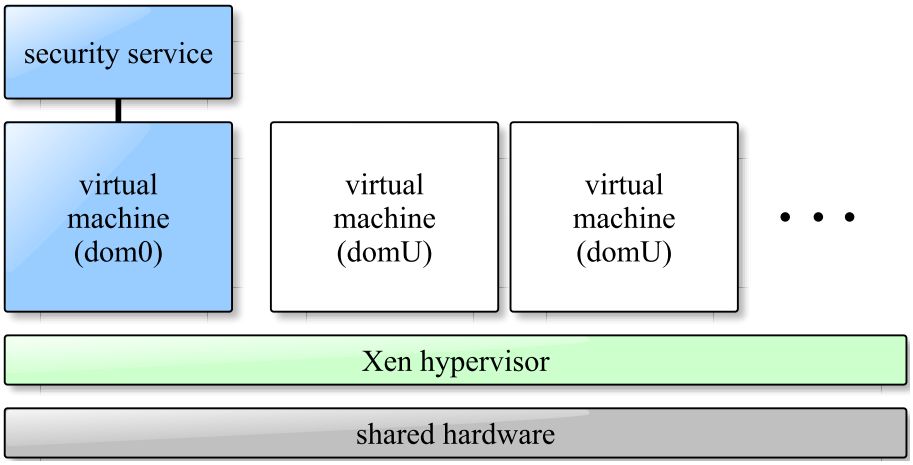

Figure 11: Location of the security service isolated and transparently operating on the administrative VM (dom0).

The service transparently retrieves the required information from different and randomly chosen VMs. In particular, a single detected anomaly is not conclusive; however, if a continuous propagation of the same anomaly can be discovered on an increasing number of different VMs within the IaaS cloud network, an autonomous propagation of unknown malware can be assumed. Our service successively scans different VMs in time intervals Δt. Once a steady increase in the occurrence of the same anomaly exceeds a defined threshold T, a signature is generated and deployed on network traffic filtering software. Figure 12 illustrates the architecture of the proposed security service which uses a centralized component called propagation detector. The detector retrieves information from the administrative VM's running on different physical servers which host multiple customer VMs in parallel. The detector transparently retrieves information about currently running processes on the system of a target VM and uses results for comparison with information in its database.

In detail, the procedure of the security service works as follows:

1. It retrieves a list of running processes from a randomly chosen customer VM by transparently investigating the corresponding volatile memory regions with VMI in combination with memory forensic techniques (Section 2.4).

2. For comparison with a white-list, it uses the name of the process or a hash over the executable code of the process (Section 4.1.1).

3. Once the service reveals an unknown running process it adds it to a list of unknown processes.

4. For each unknown process, the service scans the number of occurrences and identifies if the same unknown process continuously appears on an increasing number of customer VMs within the IaaS cloud network.

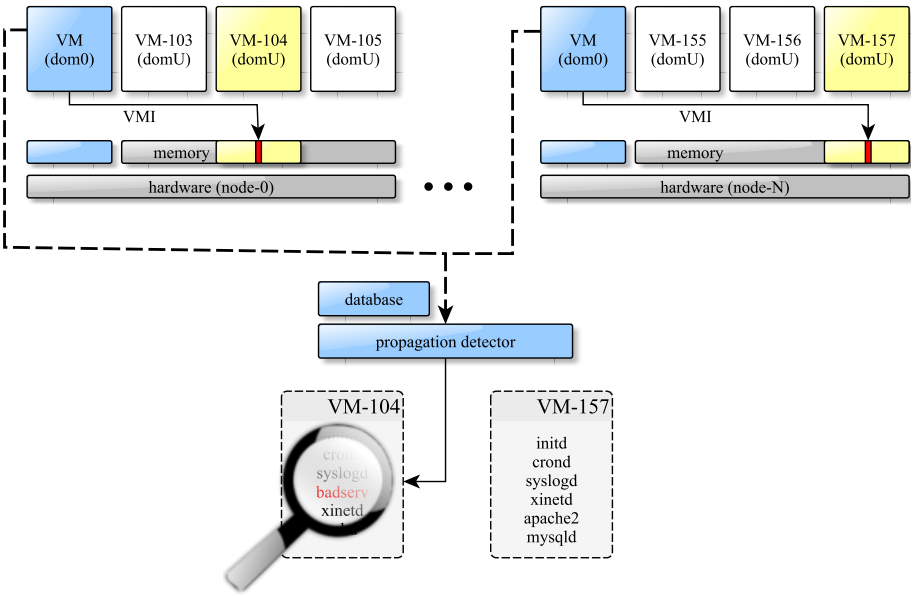

Figure 12: Architecture of the proposed security service which detects the propagation of unknown spreading malware by transparently revealing its spreading characteristics.

5. If the occurrence exceeds the threshold T, a spreading malware is found and mitigation is initialized (e.g. isolating infected VMs or deploying a signature on traffic filtering software).

6. In contrast, if the occurrence decreases without exceeding the threshold T, the process is removed from the list of unknown processes and its information is added to a white-list.

This way, unknown spreading malware can be transparently identified only by its spreading behavior itself without the need to have any previous information about the malware and without influencing or interrupting the systems of the customers. Furthermore, the proposed security service can continuously add unknown harmless processes to a white-list stored in a central database.

We implemented the proposed security service in order to evaluate the timings of the used techniques. We used a Xen hypervisor setup (Section 2.2) with an Intel Core 2 Duo CPU with 3 Ghz and 4 GB of memory. The service was requested remotely, operated on the administrative VM (dom0) and monitored a co-dissent customer VM (domU). Both VMs run Ubuntu Linux 10.04. Remotely retrieving information about running processes of the target customer VM and verifying the corresponding integrity measurements can be performed in 2.2 ± 1.8 seconds. Multiple requests can be performed in parallel retrieving information from different VMs.

Furthermore, we developed a simulation in Java in order to evaluate the proposed detection strategy in a large-scaled cloud computing scenario. We simulated the rapid propagation of malware among

different systems. In general, the detection time is influenced by the scanning interval Δt, the spreading time of the malware, a defined threshold T and the number of vulnerable VMs in the cloud network. The simulation showed that when using a scanning interval Δt of 500 ms, a threshold T of 6 and 1024 running VMs, the simulated malware having a spreading time of 2 seconds can be identified before it can affect more than 50% of the vulnerable customer systems. Once the malware could be revealed, mitigation strategies can be automatically initiated and immediately prevent further propagation. The number of infected systems greatly decreases once the malware's spreading time is higher (Table 1).

spreading time of malware	**infected customer systems until detection**
2 seconds	60%
4 seconds	42%
8 seconds	26%

Table 1: Infected systems until detection of the unknown spreading malware depending on the malware's spreading time.

The evaluation of a prototype showed the feasibility of the proposed security service. Even if the service can not prevent infection, it can mitigate the propagation and protect a notable number of customer systems. The service can be used to transparently detect unknown malware which autonomously propagates or malware which is injected with autonomously operating hacking tools ("auto-rooters") in the IaaS cloud network.

In this security service, we showed how we can benefit from a combination of properties of the IaaS cloud's underlying virtualization technologies and a broad perspective enabled by a cloud computing network in general.

In the second proposed security service, we focus on the detection of phishing attacks which are executed with the help of hitherto unknown spoofed web pages. In a recent survey about phishing attacks from 2013, Khonji et al. [42] defined phishing as "a type of computer attack that communicates socially engineered messages to humans via electronic communication channels in order to persuade them to perform certain actions for the attacker's benefit." Phishing attacks exploit the good nature of humans. Targets are found via email, web pages, SMS, VoIP or even in computer games. Phishing can lead to huge financial loss as well as loss of privacy. According to a report of the Anti-Phishing Work Group[2], from October to December 2012 more than 140.000 new unique spoofed phishing web pages have been found. About 66% targeted the financial sector and payment services. The motives of phishing attacks are mostly financial gain, but also industrial espionage, distribution of malware or harvesting passwords [90]. Even though the idea of phishing is old, phishing attacks are still a major problem since there is no automatic and effective way of elimination, mainly because it is a social-engineering attack and the human factor plays a key role. Studies [74] showed that 28% of users fell for phishing attacks, even if they have been previously trained to recognize them. Unfortunately 23% of ordinary users do not even look at the browser's address bar to verify the source of a web page [20].

Phishing attacks are social engineering attacks which aim at identity theft. In many cases, the unaware user reveals critical and sensitive information like login credentials or for example a credit card number. The most common phishing attack is redirecting a user to a prepared phishing web page which looks as similar as possible to an authentic web page of an arbitrary institution (mostly of a financial institution). With the help of the spoofed web page, the attacker requests sensitive information. In order to give a plausible scenario, faked reasons for these requests are errors in the institution's database, "verification" of the user's account or solving any problems regarding current financial transfers. Unaware users often do not recognize the ongoing attack, since they are sure to communicate with the real institution's web page. The URLs to these spoofed web pages are transmitted to the users in different ways, mostly via email including social-engineered text content to convince the user. Recently, the URLs are also transmitted via instant messaging, social networks, SMS or in Massively Multiplayer Online Games [35]. Unfortunately, an increasing appearance of specifically tailored phishing attacks can be found on the Internet. These attacks are called "spear phishing" attacks [35] because they have a specific single target and use contex-

2 http://www.antiphishing.org

tual information like the user's real name. "Spear phishing"-attacks mostly aim at high-level targets (for example CEOs) and they are much harder to detect. Since phishing is still one of the fastest growing crimes, novel and automated anti-phishing approaches are required.

In order to mitigate phishing attacks, we propose a novel anti-phishing security service which uses the underlying virtualization technologies of cloud computing and which is based on fine-grained VMI (Section 2.4). The service monitors the VMs of customers and examines the volatile memory pages that are currently assigned to the customer's browser application. Once a new web page is loaded, the service extracts certain specific information from the raw data of these memory pages and generates a unique fingerprint based on color values used in the processed web page's code as well as in buffered images and icons. Since phishing is a social engineering attack and the human factor plays a major role, the service performs human perceptual similarity analysis on the generated fingerprints in order to detect phishing attacks which are based on spoofed web pages that look similar to known authentic web pages. The proposed service generically supports every browser software in every version running in an arbitrary operating system. To the best of our knowledge, this is the first phishing detection and prevention approach based on transparent VMI techniques for a cloud computing scenario.

4.2.1 *Existing Anti-Phishing Architectures*

Contemporary browsers usually protect the users from phishing attacks with the help of black-lists which can be used in automated mechanisms. Le et al. [51] proposed the LARX architecture (Large-scale anti-phishing by Retrospective data-eXploration). LARX filters URL requests and searches for black-listed URLs retrieved from the Google Safe Browsing API[3]. However, databases which provide black-lists rely on diligent manual submitters and verifiers of suspicious URLs. Black-listing is very effective, but strongly depends on up-to-date information. New and unknown phishing web pages can remain undetected for a long period of time. Unfortunately, new phishing web pages are rapidly deployed and proxies or DNS poisoning attacks can be used to circumvent the black-lists. Automated mitigation techniques are required. Bin et al. [80] proposed a DNS based anti-phishing approach which works by sniffing the user's HTTP packets. Their architecture additionally request several trusted DNS servers for the IP address of the receiver of sensitive information and can verify if the receiver is authentic. Pal et al. [61] presented PhishBouncer, a middleware which inspects the user's HTTP traffic in order to rapidly detect and block interactions with phishing web pages. There are

3 http://code.google.com/apis/safebrowsing/

also mitigation strategies which are more offensive and suggest actively flooding identified phishing web pages with huge amounts of spoofed credentials ([91], [43]).

A lot of proposed anti-phishing architectures are extensions to the browser (plugins). These extensions monitor the web page which the browser displays and analyze certain meaningful visual characteristics. Chou et al. [17] proposed Spoofguard, a browser plugin which examines different tests on displayed web pages and summarizes the results in a score. Fotiou et al. [26] implemented a browser extension where the spoofed web pages are detected based on graphical similarity to authentic web pages. Rajalingam et al. [54] proposed a phishing detection architecture which uses key point features of specific images from full snapshots of the suspicious and the authentic web pages. They use block-wise similarity analysis of the snapshots. White et al. [88] proposed an anti-phishing architecture which uses comparison of images based on Hamming distances in combination with certain other characteristics like the number of included links. Fu et al. [28] used the Earth Mover's Distance to calculate the distances of the web page's images for visual assessment. Medvet et al. [56] identified features which play a key role in building a phishing web page that looks similar to an authentic one: Text pieces, the style, and embedded images. Rosiello et al. [71] improved the detection techniques by analyzing the entire displayed web page's DOM-tree representation.

All these anti-phishing architectures are browser extensions or software components running on the same operating system which actually means the architectures can be manipulated or disabled by malware. Even social-engineering attacks can convince unaware users and bring them to ignore or disable the protection mechanisms. If the protection mechanisms process snapshots of the browser's graphical output, they suffer from the fact that other frames can obfuscate the required features and they can be strongly influenced by frequently changing content like advertising banners. Moreover, browser extensions often support only a specific version of a browser and are quickly out of date. Our proposed anti-phishing security service is no extension to a browser, it runs isolated in the an administrative VM (dom0) and transparently extracts data from a browser's allocated memory operating on a customer VM on the same physical hardware. Furthermore, our proposed service uses human perceptual similarity analysis which takes into account that the human eye is more sensitive to certain colors than to others.

4.2.2 *Architecture of the Anti-Phishing Service*

The proposed security services focuses on the detection of phishing attacks which are based on redirection to spoofed web pages that imi-

tate authentic web pages. The security service transparently retrieves information about a cloud customer's browser by directly reading its currently allocated memory pages via VMI (Section 2.4). In comparison to the proposed security service of Section 4.1.1, in which we transparently located and verified the executable code (ELF) of a running process with the help of VMI, we now transparently analyze dynamically allocated memory pages assigned to the stack, the heap and anonymously mapped regions of a process. Figure 13 illustrates the location and perspective of the anti-phishing service operating on the administrative VM (dom0) and investigating memory pages of a customer VM. Figure 14 illustrates the different memory layers investigated by the proposed service. First, it retrieves the physical memory region allocated by the target customer VM, then, depending on the used operating system, it investigates specific static kernel information in order to retrieve the addresses of the currently allocated memory pages of the browser process. Finally, the located memory pages of the process in the user space are read and the stored information is analyzed.

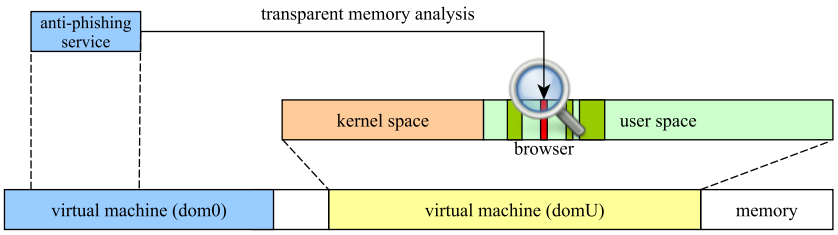

Figure 13: Perspective of the proposed anti-phishing security service.

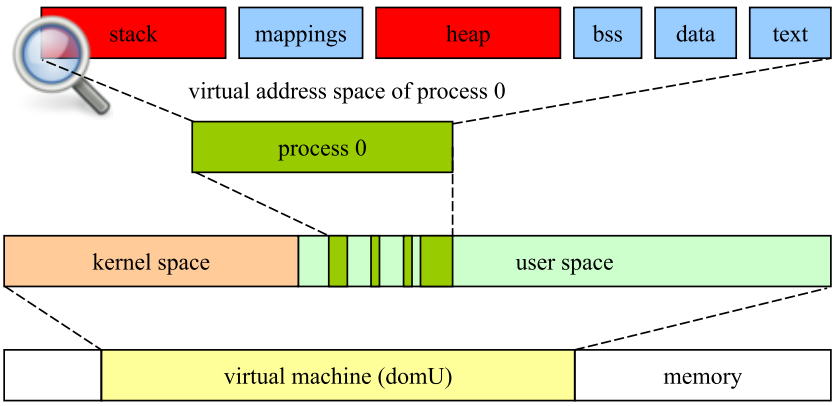

Figure 14: Different memory regions investigated in order to analyze dynamically stored data of a target process (browser application) from the transparent hypervisor perspective.

The operation of the anti-phishing service is illustrated in a flow graph having different sequential steps running in a loop (Figure 15). In general, there are two major components: The *extractor* which lo-

cates, retrieves and pre-processes the required information from the raw data read from the volatile memory and the *detector* which performs human perceptual similarity analysis and intervenes once an ongoing phishing attack was detected.

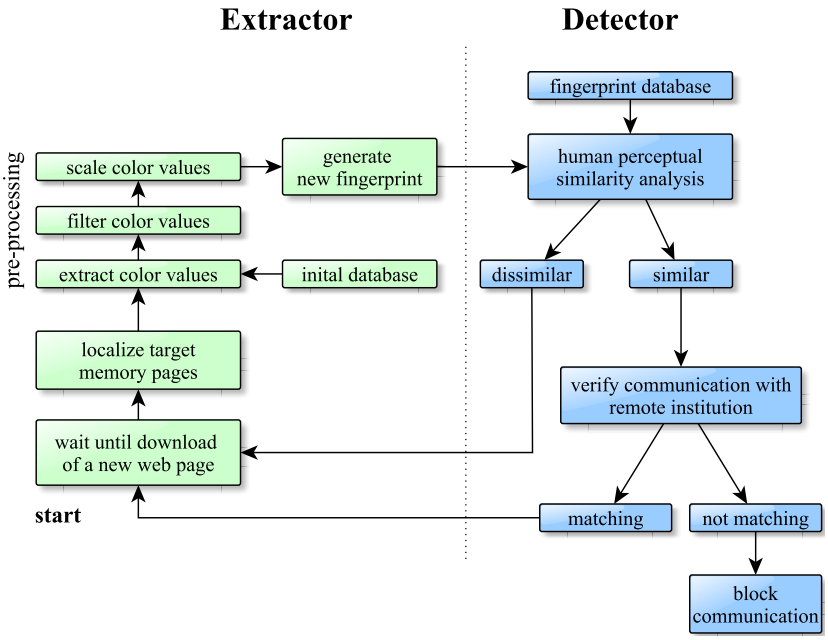

Figure 15: Flow-graph of the proposed anti-phishing security service.

4.2.2.1 *Design of the Extractor*

The extractor waits for the download of new web content. Once it could identify a new download, it generates a new fingerprint from a web page which the customer's browser currently processes. The extractor is responsible for localizing, retrieving, filtering and balancing color values from different sources present in the accessed and processed web page into a fixed-length fingerprint. Since contemporary browsers use a lot of memory to store and process various information, the performance of the extractor is an important issue. The extractor is triggered once new incoming HTTP traffic to the customer's VM from a new remote source could be detected. The network traffic of the VM can be transparently monitored and analyzed since it passes the hypervisor layer which can be accessed by the extractor that operates in the administrative VM (dom0). Once an extraction procedure is triggered, four subsequent steps are executed:

1. **Localization:** Memory pages which the browser has currently allocated and which are assigned to the stack or the heap are identified. Buffered images and the code of web pages are localized within these identified memory pages.

2. **Extraction:** Color values are extracted from the localized content while content belonging to the browser application is ignored. This is done based on comparison of the content to an initial database containing typical colors of the browser software.

3. **Filtering:** Extracted color values which are not characteristic enough and which belong to known false-positives are removed.

4. **Scaling:** Since the extractor uses two different sources to retrieve color values (buffered images and the web page's code), the collected values are balanced and scaled into a fingerprint having a fixed length.

4.2.2.2 *Design of the Detector*

The detector processes the newly generated fingerprint from the extractor to reveal an ongoing phishing attack. The detector performs human perceptual similarity analysis on the new fingerprint and stored fingerprints of known authentic web pages from its database. The fingerprints of this database are previously generated from a number of popular authentic web pages (including their IP addresses) which are known to be a frequent target of phishing attacks (for example popular financial institutions). Similarity analysis is performed by calculating the human perceptual color distance of the new fingerprint to each one of the known fingerprints from the database. Once a result shows high similarity between two fingerprints, the detector inspects the current network communication of the customer's VM in order to match the identified institution to its valid known remote IP address. If a download of a web page from the corresponding remote source could be verified, there is no need to intervene and the currently processed and displayed web page was downloaded from the authentic institution. On the contrary, if there was no communication with the corresponding remote source on the network, but the web page which is processed and displayed on the browser has a high similarity to a web page of a known institution, an ongoing phishing attack based on a spoofed web page is assumed. In this case, the cloud customer is redirected to a prepared web page on which information about the incident is displayed.

The proposed security service is able to detect unknown phishing pages which have never been seen and investigated before. Our strategy brings the creators of phishing pages into a dilemma: The more similar a prepared phishing page is to the original web page, the easier it can be detected by our service. However, if phishing pages are created which look that less similar to the original pages so that our security service will not detect them, the users may not fall for the phishing attack anyway.

4.2.3 *Implementation and Strategy*

We implemented the security service for the Xen hypervisor (Section 2.2) and used fine-grained VMI (Section 2.4) for live memory forensics. In comparison to the previously proposed security service used for malware detection (Section 4.1.1), we now transparently investigate dynamic memory regions.

4.2.3.1 *Implementation of the Extractor*

The extractor detects an ongoing download of a new web page from a new remote IP address by monitoring the incoming HTTP to the target VM. In order to extract only relevant information and no browser internal data, the extractor maintains a list of hashes from a browser's internally used images (for example icons) as well as internally used HTML code segments (for example of help pages). By comparing new computed hashes over the extracted content of the current scan and hashes stored in a database, the extractor can reveal new and relevant content which is used for further analysis. The extractor creates a vector of independently extracted RGB values as a fingerprint which is characteristic for a currently processed and displayed web page. For this purpose, the extractor identifies the locations of images and icons in the memory based on specific unique start and end byte sequences of different formats (e.g. PNG and JPEG). From each localized image, the extractor rapidly extracts s randomly selected RGB values. For each image, the number s of extracted random colors is calculated with an optional initial parameter L_s by $s = (width_{img} + height_{img}) \cdot L_s$ where width and height are the dimensions of the image. This way, larger images influence the resulting fingerprint more than smaller images. Furthermore, randomly selecting RGB values leads to the establishment of characteristic RGB values. The extracted RGB values of the images are stored in a temporary list.

Images can use a big amount of white and black color values which often belong to letters in text content or just to the background. In general, web pages use a lot of very bright subtypes of white colors and very dark subtypes of black colors which the human eye cannot really distinguish. These RGB values are not characteristic for a web page. The extractor calculates the brightness b of each RGB value by $b = \sqrt{\frac{r^2+g^2+b^2}{3}}$ and drops RGB values which have a brightness below an optional limiting parameter L_{bottom} or which have a brightness exceeding an optional limiting parameter L_{top}. This way, very bright and very dark RGB color values are dropped.

In order to extract a characteristic color-based fingerprint of the currently processed web page, the extractor focuses on characteristic colors, for example of logos and icons which often have a unique style. In particular, logos or icons use less different colors than for

example photographs. This means these images are usually characterized by the fact that the same RGB values appear more often in the amount of selected random pixels. Since a web page can contain photographs which are less characteristic for the web page, the extractor filters the list of RGB values depending on the occurrence of each extracted RGB value. RGB values which occur less frequently than a previously defined optional limit L_{occ} are dropped. Finally, the extractor has created a filtered list of RGB values of the browser's processed images.

After processing the color values from buffered images and icons, the extractor localizes HTML code snippets within the analyzed memory pages and extracts color values in hexadecimal form with the help of regular expressions. These HTML color values are also temporarily stored in a list which is subsequently filtered. Since the regular expressions can match false-positives which have the same spelling than hexadecimal colors but are actually no real color values (for example #dec is often used as a "decode" tag but also matches a green color shade in hexadecimal), these known false-positives are additionally filtered. Subsequently, the HTML hexadecimal color values are converted into RGB color values and they are also filtered based on their brightness and their occurrence like the RGB values of images.

Since there are usually much more RGB values of images than of the HTML code, but the HTML RGB color values are very characteristic for a web page (for example they are used as font colors or as the background color of an entire web page), the lists of RGB values from the two different sources are merged depending on a weighted optional balancing factor f_b. Following this strategy, the extractor adds HTML color values n-times to the list of both types where $n = \frac{colors_{images} \cdot f_b}{colors_{html}}$. Finally, the extractor creates a weighted fixed-size color fingerprint vector of the length f_s. In summary, the extractor's procedures and a resulting fingerprint depends on the parameters listed in Table 2.

parameter	description
L_s	number of selected random RGB values from images
L_{bottom}	brightness limit for RGB values (bottom)
L_{top}	brightness limit for RGB values (top)
L_{occ}	minimum required occurrence for RGB values
f_b	balancing factor for the merging procedure
f_s	fixed size of the resulting fingerprint

Table 2: Parameters influencing the creation of a color-based fingerprint.

Since the browser can process different web pages or multiple tabs of the browser can be used, the extractor maintains a list of hashes over sets of already analyzed memory pages. Once a download of a new web page could be detected, the extractor can calculate new

hashes for comparison and only analyze memory pages which have been modified (and most likely contain the new processed data).

4.2.3.2 *Implementation of the Detector*

The detector compares a new generated fingerprint with known fingerprints from a database. Comparison of two fingerprints is performed with the help of human perceptual similarity analysis of colors which closely resembles the way a human recognizes colors and their differences. Since phishing attacks are social engineering attacks, analyzing the similarity in a human perceptual view brings several advantages. Small changes in the used colors on a web page do not necessarily lead to changes in the human perception of that page and the human eye is more sensitive to certain colors than to others. In particular, the "look-and-feel" of the characteristic design of the web page as a whole is compared.

The similarity analysis is performed as follows: First, the detector converts the RGB color values of the fingerprints into the CIE[4] Lab color space [83]. The Lab color space (L*a*b) uses luminance (L) and two color channels (a, b). For the conversion, the detector uses an observer in 2° and the CIE Standard Illuminant D65. In the L*a*b* color space, changes in a color approximately correspond to the human perception and therefore it is particularly suitable for our purposes.

Given two colors (L_1,a_1,b_1) and (L_2,a_2,b_2) and the application depending weighting factors $S_C = 1 + 0.045 * (a_1^2 + b_1^2)$ and $S_H = 1 + 0.015 * (a_1^2 + b_1^2)$ (recommended factors by CIE for graphic arts) and Equation 1 and Equation 2, the detector can compute the human perceptual color distance Δe of the two colors by applying the CIE (94) formula for color distance (Equation 3).

$$\Delta C = \sqrt{a_1^2 + b_1^2} - \sqrt{{a_2}^2 + {b_2}^2} \tag{1}$$

$$\Delta H = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 - (\Delta C)^2} \tag{2}$$

$$\Delta e = \sqrt{(L_1 - L_2)^2 \cdot \left(\frac{\Delta C}{S_C}\right)^2 \cdot \left(\frac{\Delta H}{S_H}\right)^2} \tag{3}$$

The detector performs similarity analysis of two fingerprints F_1 and F_2 by sorting the color values and computing the perceptual distances of each color value from F_1 to the color value on the same position in F_2. This yields a vector of individual distances. Finally, the detector computes the length of the difference vector with the Euclidean norm.

4 International Commission on Illumination (CIE)

If a new fingerprint from the browser's memory and a fingerprint from the database are very similar, but there was no communication with the known remote IP address of the corresponding institution, an ongoing phishing attack is assumed.

4.2.4 *Evaluation*

For an evaluation, we used an Intel Core 2 Duo CPU with 3 Ghz, 4 GB of memory, a Xen hypervisor (4.1) setup and a customer VM with Ubuntu Linux 12.04 running Mozilla Firefox. We investigated how characteristic the generated fingerprints are and how feasible the proposed anti-phishing service is under realistic conditions.

4.2.4.1 *Timings and Fingerprint Generation*

First, we evaluated the timings of the procedures. For this, we measured the timings of three sub-procedures using the parameters L_s of 4, L_{occ} of 4, L_{bottom} of 3, L_{top} of 252, f_b of 0.5 and f_s of 64. Based on 100 independent test-runs, the color value extraction and processing takes 2.6 ± 0.2 seconds, the fingerprint generation 1.2 ± 0.1 seconds and a similarity analysis 0.2 ± 0.1 seconds in our setup. On average, the proposed service requires around 4 seconds to scan the memory pages of the browser of the target customer's VM, to extract the required data, to generate a new, filtered, balanced and scaled color-based fingerprint and finally to analyze the perceptual difference of two different fingerprints. This is usually fast enough to detect an ongoing phishing attack before an ordinary user can enter valid credentials on a real phishing web page. However, we nevertheless use Linux HTTP outgoing traffic shaping on the hypervisor layer to delay the outgoing traffic of the VM until the analysis is finished.

The generated fingerprints can often be distinguished just by using the human eye. Figure 16 illustrates two example fingerprints from the Chase bank (left) and the ICICI bank (right) in visualized color maps (f_s of 64).

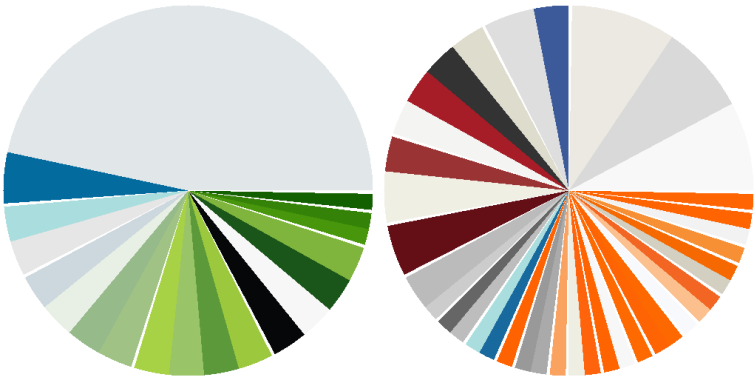

Figure 16: Fingerprints of the Chase and the ICICI financial institutions.

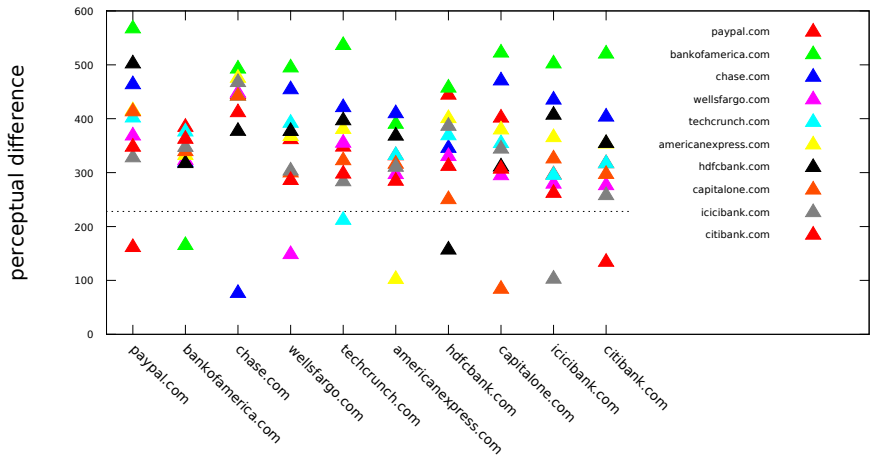

Figure 17: Differences as result of human perceptual similarity analyses of generated fingerprints among the Alexa top 10 web pages in the category financial services. Each triangle represents the average of 100 analyses with new generated fingerprints. Analyses of fingerprints of the same web page have always the lowest difference.

Characteristic colors which belong to the specific brands can be found and they can be identified on both web pages. The web page of the Chase bank prefers different shades in very light blue and green while the web page of the ICICI bank uses shades in light gray, red and orange. There are several characteristic colors hidden on the web pages which can be seen during interaction with the web page, for example if menu buttons are clicked.

For automated evaluations, we used the Alexa top 100 web pages in the category financial services[5]. These institutions are continuously top targets of phishing attacks. In general, the generated color-based fingerprints slightly vary, although they are generated from the data of the same web page. This is because the extractor randomly selects RGB values of localized images. In particular, usable fingerprints generated from the processed memory pages need to have two major requirements: A stable low difference between two fingerprints of the same web page and a high difference between fingerprints of different web pages. Figure 17 shows results of the human perceptual similarity analysis among the created fingerprints of the top 10 institutions, in each case the average of 100 independently generated fingerprints. The lower the resulting perceptual difference, the more equal are the color-based fingerprints in a human perceptual way.

In Figure 17, the difference between the fingerprints is always the smallest if both fingerprints are generated from the same web page. Two fingerprints can be clearly distinguished if they are generated from web pages of two different institutions. In these cases, the human perceptual difference is much higher. In our test-runs, a common

5 http://www.alexa.com/topsites/category/Top/Business/Financial_Services

threshold which could be used to detect real phishing pages targeting these 10 web pages could be 228. We also evaluated fingerprints of all Alexa top 100 web pages in the category financial services and compared them to each other. On average, the perceptual difference among fingerprints of a target web page and fingerprints of all the other web pages is 372.1 ± 21.7.

A false-positive occurs if two fingerprints of two different web pages cannot be distinguished because the perceptual difference between them is equal or even less than the perceptual difference between two fingerprints from the same web page. This case occurred only once in our test-runs which leads to a false-positive rate of 1%. Table 3 gives an overview of the average number of processed color values during the fingerprint extraction procedures of the Alexa top 100 web pages.

description	**average number**
RGB values extracted from images	6766 ± 8276
RGB values extracted from images after filtering	2504 ± 2017
RGB values extracted from HTML code	392 ± 218
RGB values extracted from HTML code after filtering	383 ± 213

Table 3: Number of processed RGB values on average before scaling into a weighted fixed-size fingerprint (they include repetitive values).

In particular, the color-based fingerprints generated from the customer's volatile memory and the results of the human perceptual similarity analysis performed on these fingerprints are characteristic enough to be used in our transparent anti-phishing service. Fingerprints of selected institutions and their average difference among each other can be stored in a database. If a computed perceptual difference of a currently processed web page falls below a defined limit, our service can investigate the traffic and trigger an alert. With the help of the fingerprints, the proposed service can reliably reveal if a spoofed and processed web page looks similar to an authentic web page. The more a spoofed web page is actually perceived like an authentic web page in a human perceptual way, the better it can be detected.

We investigated the dependence of the similarity analysis of the Alexa top 10 on the parameters L_{bottom}, L_{top}, L_{occ} and L_s. The higher the average perceptual difference among the fingerprints, the better the used parameters are for our purposes. Fig 18 shows a clear trend when using different values for L_{bottom} and L_{top}. The more colors are removed because of their brightness or their darkness the less the fingerprints differ. Accordingly, color values should only be removed if they are very bright or very dark. The more colors are removed depending on how often they occur (L_{occ}), the more different are the fingerprints (Fig 19). Consequently, the more often a color oc-

curs, the more characteristic it is. Furthermore, the number of pixels which are randomly chosen from located images should not be too small to get an overview on the characteristics (L_s), but also not too high since it slows down the procedures (Fig 20). As robust values for the parameters, we suggest to use a L_{bottom} of 4 and a L_{top} of 251 for brightness and darkness filtering, a L_{occ} of 4 for filtering based on the occurrence and a L_s of 4 for the number of selected random pixels from each image.

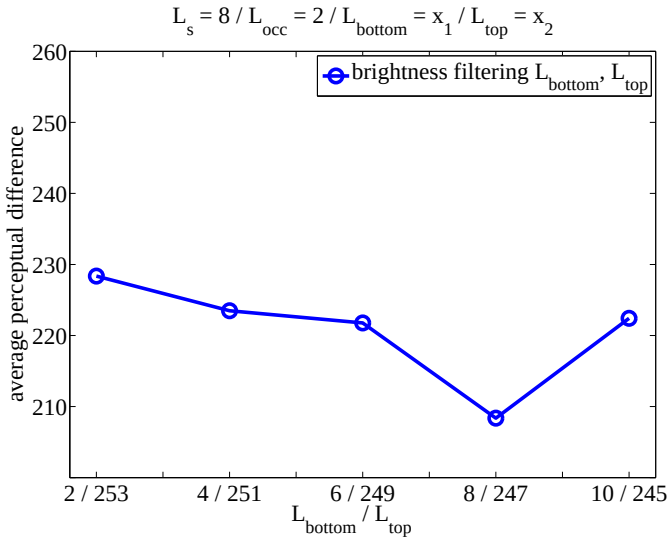

Figure 18: Average difference in the human perceptual similarity analysis depending on the parameters for brightness filtering (L_{bottom} and L_{top}).

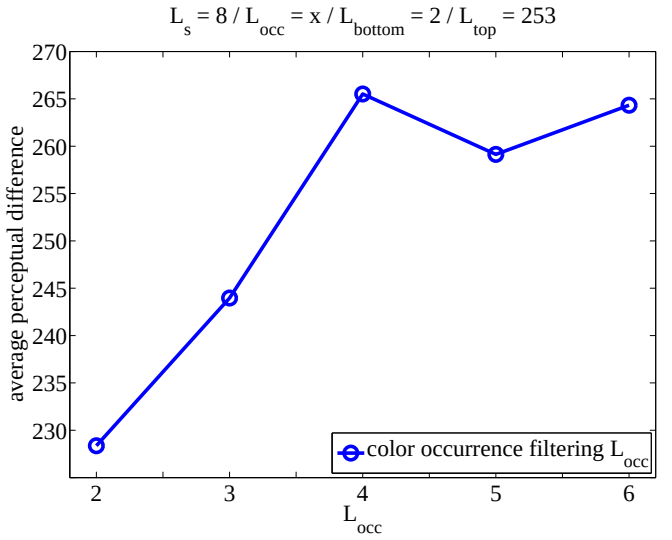

Figure 19: Average difference in the human perceptual similarity analysis depending on the parameter for occurrence filtering (L_{occ}).

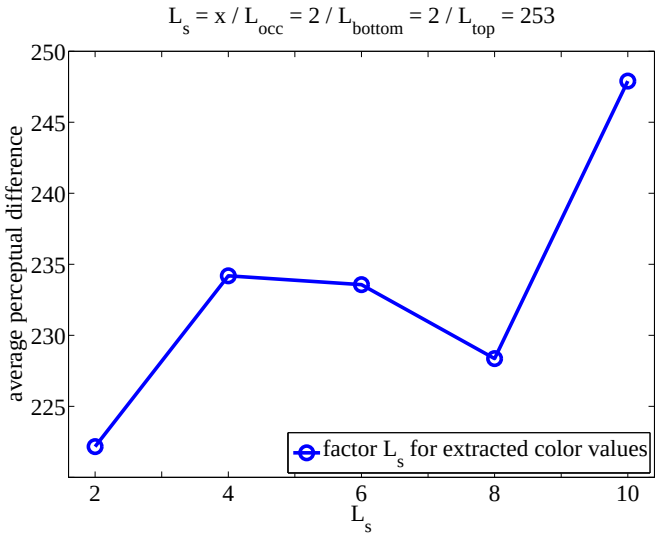

Figure 20: Average difference in the human perceptual similarity analysis depending on the parameter for the number of selected pixels from buffered images (L_s).

4.2.4.2 *Evaluation with Real Phishing Web Pages*

Furthermore we investigated whether real spoofed web pages look similar to the authentic web pages of the target institutions by using our proposed fingerprint comparison approach. For this evaluation, we used real phishing web pages which we could currently find on the Internet[6]. Figure 21 shows the human perceptual differences among original web pages (green) and among the original and real phishing web pages (red). The results clearly show that the real phishing web pages are strongly similar to the original web pages based on our generated color-based fingerprints from the raw data transparently read from the volatile memory. All of them are located under the exemplary threshold of 228 (Figure 17).

On average, the difference between a fingerprint of an original and a corresponding phishing web page is 51.2 ± 16.3, which is much smaller than the average difference of the fingerprints between a selected web page and all other web pages of the Alexa top 100 which is 372.1 ± 21.7. The perceptual similarity analysis of original and real phishing web pages show that our approach is feasible.

6 http://www.phishtank.com

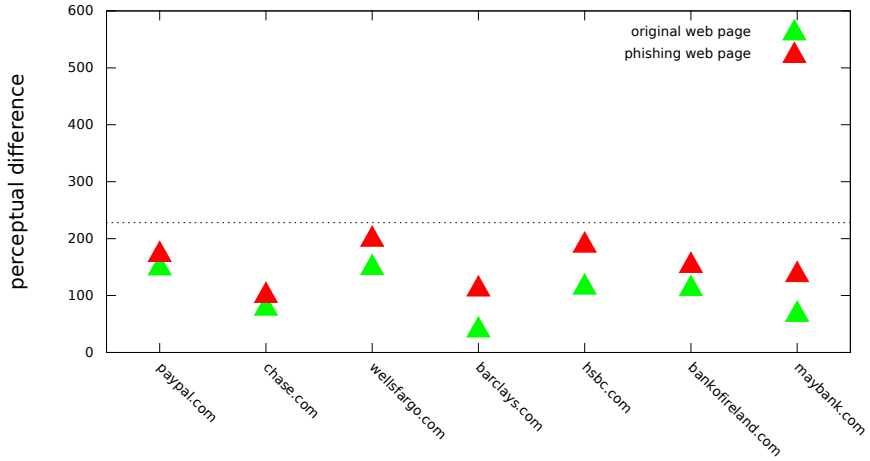

Figure 21: Perceptual differences of fingerprints among original web pages (green) and of fingerprints among the original and real corresponding phishing web pages (red) (average of 100 independently generated fingerprints).

The proposed anti-phishing security service detects ongoing phishing with the help of our developed techniques and strategies and reveals spoofed web pages that look similar to original web pages of authentic institutions.

The third proposed security service focuses on a different type of phishing attack which is executed with the help of malware that successfully infected the customer's operating system. Compromising the operating system with malware is often done by exploiting software vulnerabilities. However, in the most cases a user is just tricked to download and install some software additionally containing malware components. Once the malware is installed, it can execute arbitrary operations on the customer's software platform. Some malware focuses on phishing attacks and injects malicious content into the code of authentic web pages which the browser processes before it is displayed. This kind of phishing attack is called "Man-in-the-Browser"-attack (MitB) [84] and has several advantages in comparison to phishing attacks based on redirection to spoofed web pages. MitB-attacks use malicious browser extensions [68] and modify the web page's code while it is processed by the browser on-the-fly. An authentic web page is modified after it is downloaded but before it is displayed to the user. This way, web forms can be replaced and detection mechanisms are bypassed. Sensitive information which the user enters can be captured before it is encrypted and sent to the real remote institution, even after successful authentication on the institution's original web page. For example, the sophisticated third-generation malware SpyEye [76], which is the successor of the famous Botnet client Zeus [13], uses MitB techniques. Our third proposed security service in the field of transparent attack detection reveals and prevents "Man-in-the-Browser"-attacks.

4.3.1 *Architecture of the Anti-MitB Service*

A MitB attack in our IaaS cloud computing scenario is illustrated in Figure 22. Malware has already infected the operating system of a customer's VM. While the customer downloads an authentic web page, the malware modifies the web page's code on the application layer while it is processed by the browser but before it is displayed. However, our proposed MitB-detection service, which operates on the co-located administrative VM (dom0), captures the authentic web page's code during its transfer through the virtualized network device in the Xen hypervisor setup (Section 2.2) before the MitB attack is executed.

In order to detect an ongoing MitB attack, the service retrieves information about the currently processed web page from the volatile memory as well as it investigates the captured HTTP traffic. The service performs similarity analysis of the web page which is downloaded and the web page which is finally processed and displayed by the browser. To identify ongoing MitB attacks, the detector can

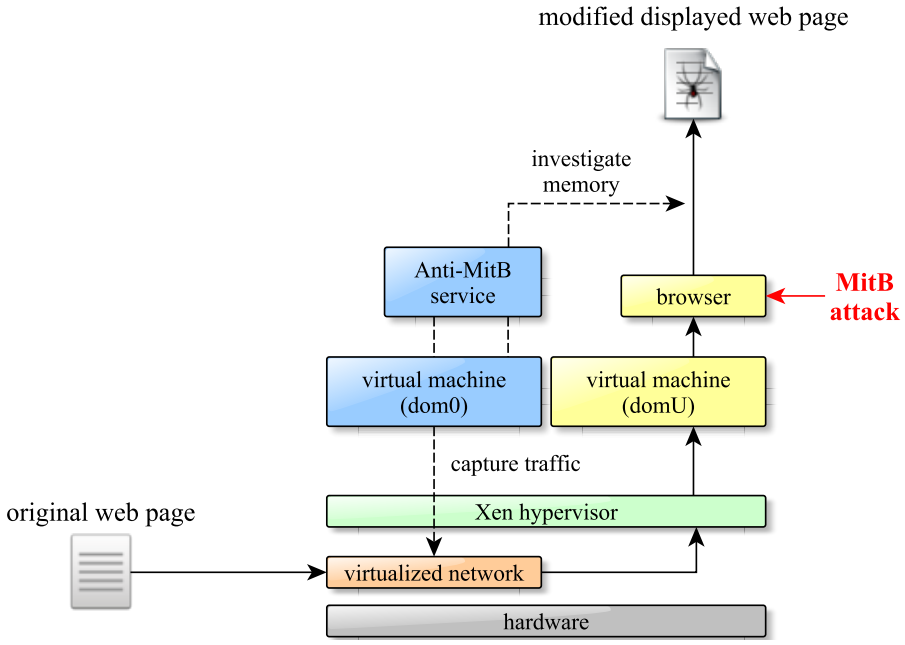

Figure 22: MitB attack: The original web page passes the hypervisor and is modified at the application layer on the customer's VM before it is displayed. The detection service can retrieve information about the web page before and after the browser has processed it.

again use color values for similarity analysis (Section 4.2), but better results are achieved if it analyzes and compares other characteristic content of a web page's code in order to reveal modifications. Our MitB-detection service generates a fingerprint from the captured HTTP traffic of the downloaded web page and it compares the fingerprint to a second fingerprint transparently retrieved via VMI from the browser's allocated memory pages. The detector simply counts and compares the occurrence of certain characteristic elements. For example, it counts the number of <form> HTML tags in the code that is transferred to the customer's VM and in the data that is processed and stored by the browser in the volatile memory. This way, the security service can detect modifications in the processed HTML code, for example caused by the insertion of a new web form. If the transfer of the web page to the customer's VM was encrypted (SSL/TLS), the service can request a new download from the same source in order to generate a fingerprint from the unmodified content.

4.3.2 *Evaluation with Real Malware*

For the evaluation of the MitB-detection service, we use a VM that runs Windows Server 2008 R2 SP1 and the Mozilla Firefox. We intentionally infected the system with the PUP.Optional.Walermis and PUP.Optional.Desk365 malware. This malware uses MitB techniques

to redirect search queries, to inject additional advertising banners into displayed web pages and it is also known for phishing attacks.

Figure 23 shows the number of HTML snippets (HTML code identified by a start and end HTML tag) the browser processes. The infected browser clearly processed more snippets while visiting bankofamerica.com or chase.com, which means additional unknown content was inserted into the web page on the application layer.

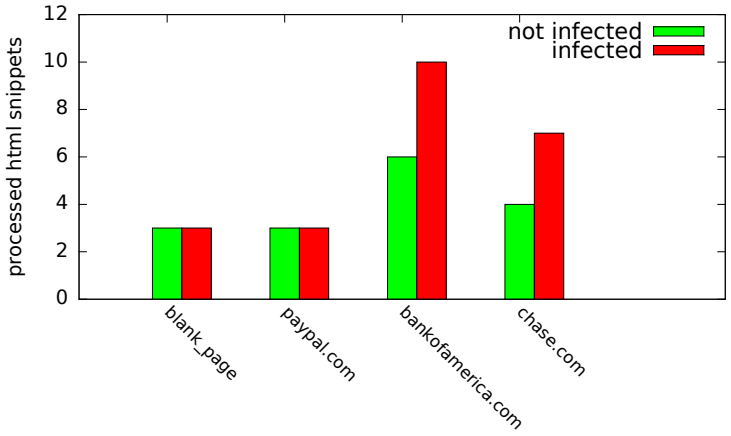

Figure 23: Processed HTML snippets during visiting a blank page and the Alexa top 3 financial services, both with an unmodified browser and the malware-infected browser.

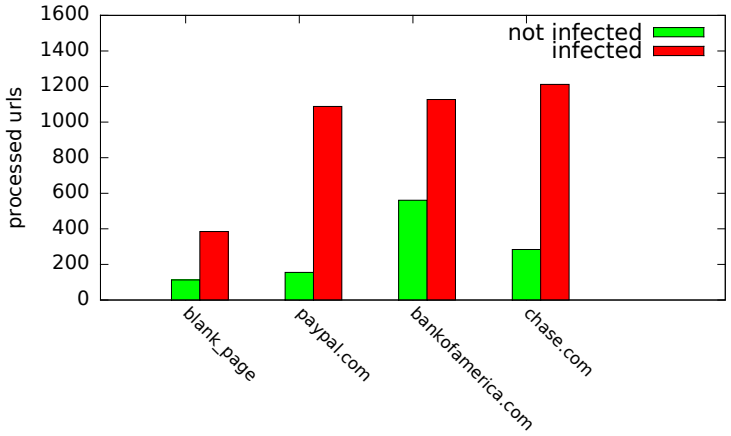

Figure 24: Processed URLs during visiting a blank page and the top 3 financial services, both with an unmodified browser and a malware-infected browser.

The difference becomes obvious if the service compares the URLs a browser processes. In Figure 24, it can be seen that the infected browser processes two times more URLs even on the blank start page which is presumably caused by the malware components performing update requests or communicating with malicious targets. Once the infected browser processes a web page of the financial targets, it pro-

cesses up to six times more URLs than it does in the setup that is not infected. Our security service can store the average number of HTML snippets and URLs a browser which is not infected processes for a target known web page and detect ongoing MitB attacks on the customer's VM once previously defined thresholds are clearly exceeded.

4.4 SUMMARY

In this chapter, we proposed three different novel security services for attack detection in an IaaS cloud computing scenario (Section 1.4) which follow the Security-as-a-Service business model (Section 1.5). The first service transparently performs integrity measurements on processes which run in the VMs of cloud customers and uses a perspective over the entire cloud network in order to identify the propagation of unknown malware based on its spreading characteristics (Section 4.1). The second security service uses fine-grained VMI (Section 2.4) to extract, filter, balance and scale a color-based fingerprint from the raw data of a web page which the browser of a cloud customer currently processes in its allocated memory pages (Section 4.2). The service uses human perceptual similarity analysis to compare generated fingerprints in order to reveal ongoing phishing attacks which are based on spoofed web pages that look similar to known authentic web pages in a human perceptual way. The service detects phishing attacks which using hitherto unknown phishing web pages that have never been analyzed before. The service operates generically on the raw memory and this way it supports every browser application in every version. Finally, the third security service transparently detects "Man-in-the-Browser"-attacks by comparing characteristic segments of the content of a web page which a browser processes and the original downloaded content of the same web page which passed the hypervisor layer (Section 4.3). These three proposed security services run isolated, they cannot be manipulated or tricked by malware and they can be flexibly provided according to the paradigm of "pay-per-use". They can easily be enabled by a cloud customer since no additional software needs to be installed or configured on a customer VM.

5 TRANSPARENT ATTACK ANALYSIS

In this chapter, we introduce two different transparent security services which analyze ongoing attacks executed against the VMs of the cloud customers. The first security service dynamically extracts virtual honeypots from customer VMs in order to analyze the strategies of attackers and to reveal vulnerabilities. The extracted honeypots run exactly the same software in the same configuration like the original VMs being under attack, but they are restricted and they do not allow access to any sensitive data (Section 5.1). The second proposed security service traps detected attackers on the process level in an isolated environment in order to analyze strategies and intentions (Section 5.2).

5.1 DYNAMIC HONEYPOT EXTRACTION

In this section, we propose a dynamic honeypot service for the IaaS cloud. Honeypots allow to detect unknown attacks and to study new attack procedures and patterns. Honeypots are provided either as low-interactive systems analyzing the incoming network traffic or as high-interactive systems allowing attackers to access the system in order to analyze their strategies in more detail. In the IaaS cloud scenario (Section 1.4), we use the hypervisor layer for precise monitoring, for example the virtualized network device for transparent network monitoring. Furthermore, we benefit from the fact that new VMs can be quickly deployed, migrated, cloned or destroyed (Section 2.3).

This flexibility brings several opportunities which we use to improve the design of traditional honeypot architectures. We detect attacks against the customer VMs in their initial phases and promptly redirect the attacking source to an instantly deployed honeypot without arousing suspicion of the attacker and without terminating the procedures of executed attacks. In this section, we report the design, the implementation and evaluation of a fast modifying VM live-cloning architecture which extracts a reduced VM from an attacked customer VM. We use this dynamically extracted VM as a honeypot VM. The proposed method has the advantage of redirecting remote attacks to a VM which provides exactly the same software components in exactly the same up-to-date configuration like the original attacked VM, but it does not allow access to any sensitive data nor it does risk the integrity of the original VM. Our proposed honeypot service can finally investigate if an attack would have been successful on the original VM and immediately inform the customer about

potential vulnerabilities or misconfigurations. We demonstrate that timely extraction and precise monitoring can be profitably used to deploy and benefit from temporary extracted honeypots in the IaaS cloud scenario. In Section 5.1.2, we describe the dynamic extraction of low-interactive honeypots, in Section 5.1.3 we describe the extraction of high-interactive honeypots.

5.1.1 *Existing Honeypot Architectures*

The concept of honeypots was first described by Clifford Stoll in his book "The Cuckoo's Egg" in 1990. Some years later, in 1997, first honeypot projects appeared and in 1998 the first commercial honeypot ("CyberCop Sting") was available. This honeypot already used multiple virtualized systems hosted on a single hardware component. In 1999, the Honeynet[1] project was founded. In this project, a group of security researchers worked on the development of honeypots and they defined the goals of honeypots as "To learn the tools, tactics and motives involved in computer and network attacks, and share the lessons learned". Since then, the development of sophisticated honeypots as well as honeynets continued [66]. In recent years, honeypots have become steadily more scalable and flexible. Particularly well-known and widely-used is the scalable honeyd project of Niels Provos [65]. Honeyd can create a number of virtual hosts on a network and can flexibly be configured to emulate arbitrary services. In particular, the timely detection and redirection of malicious network traffic to dynamically deployed honeypots is constantly a topic of interest and research. Chen et al. [16] used the redirection approach and a dynamic forensic system for specific investigations in order to minimize false-positives. In their work, network traffic which seems to be anomalous is redirected to shadow servers and a forensic module collects useful information about executed attacks.

In general, the idea of deploying dynamic honeypots instead of maintaining static honeypots was first discussed in [15]. Sardana et al. [72] suggested to use dynamic honeypots to adapt and protect a network which is under a distributed denial of service attack. Kuwatly et al. [46] proposed a dynamic honeypot system which adapts itself to the current network surroundings by passively observing the network traffic. This approach helps to automatically fit the honeypot seamlessly into the network in which it operates. Hecker et al. [34] improved this idea by adapting honeypots depending on other hosts in the local network by using active port scans instead of passive traffic analysis. This way, the dynamic honeypot can autonomously and rapidly integrate into a continuously changing network which is especially interesting for the IaaS cloud scenario being subject to continuous changes. Artail et al. [5] first deployed virtual low-interactive

1 http://www.honeynet.org

honeypots which emulate services and which redirect malicious traffic to virtual high-interactive honeypots. All these approaches deal with the autonomous and dynamic integration of honeypots in constantly changing network surroundings.

The cloud offers several advantages to design honeypots and also to deliver collected results as a security service to the cloud customers [7]. There are some honeypot architectures which use VMI (Section 2.4) or which use VM live cloning techniques (Section 2.3). Vrable et al. [85] deployed a large virtual honeyfarm called "Potemkin" based on virtualization technologies and extensive cloning. There are also approaches to deploy large-scale on-demand high-interactive virtual honeypots [44]. Lengyel et al. [49] presented "VMI-Honeymoon" and used fine-grained VMI to precisely monitor a honeypot VM. They extended their virtual honeypot architecture to a scalable honeynet system with automated malware capturing [50].

In the work of this section, we especially benefit from the scalability of virtualization technologies and the IaaS cloud scenario (Section 2.3). Our proposed honeypot security service is based on VMI (Section 2.4) and VM live cloning [81] derived from a modified VM live migration procedure [18].

5.1.2 *Low-interactive Honeypots for Attack Evasion*

In this section, we describe the architecture of a honeypot service which dynamically extracts a *low-interactive* honeypot from a customer VM. First, we analyze current network scans and attacks from real-world data and use the results to design the service.

5.1.2.1 *Analysis of Network Attacks*

We analyzed captured network traffic in order to gain an insight into current malicious network traffic. Depending on the targets and strategies of executed attacks, we design the honeypot service. In this context, we do not investigate the sources of attacks, but we focus on the frequency of network attacks and how they are executed in detail. Depending on the strategies of the most frequent network attacks, we finally design the dynamic low-interactive honeypot service. First, we analyzed four weeks (January 2012) of network TCP traffic in a large distributed darknet [1] (also known as network telescope or black hole network). The darknet captures network traffic which targets IP addresses being not assigned to a host. Figure 25 shows the network services which are most frequently contacted by unknown remote hosts. These scans are usually performed by automated scanning tools. The most scanned service is the Microsoft directory server. Attackers scan for misconfigurations and try to discover accidentally exposed data. In this section, we focus on network attacks targeting

Linux. In this context, the most connection attempts are to the Web server (80), TELNET (23), X11 (6000) and SSH (22).

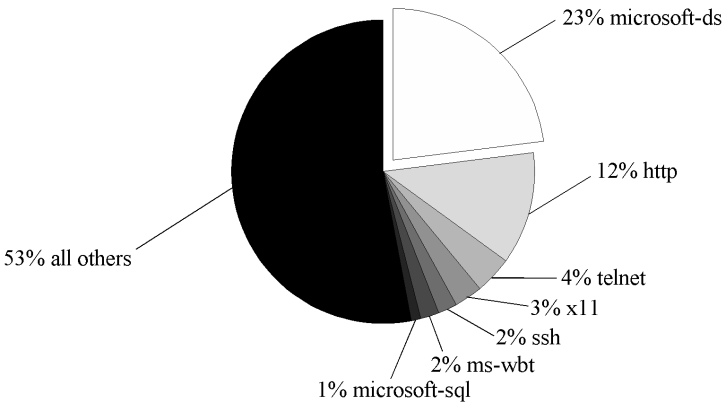

Figure 25: TCP services with the most connection attempts in a darknet in January 2012.

Next, we analyzed four weeks of captured traffic (February 2012) with the intrusion detection tool Snort[2]. Results showed that most packets are generated by ICMP requests. This can be explained with activity belonging to the reconnaissance phase of autonomous attacking tools or computer worms (Table 4). Attacking tools try to identify hosts in the network which are available before deeper scans or specific attacks are initiated. Results also showed a lot of SSH brute force or dictionary attack attempts, which means the same source repeatedly tries to contact the SSH port of the same IP address. Furthermore, another result is the significant occurrence of SQL worm activity massively trying to send payloads to remote SQL servers.

threat class	**occurrence**
ICMP ping requests	56%
SSH brute force attempts	28%
SQL worm propagation	11%
portscans and portsweeps	3%
others (tcp port 0 traffic, etc)	2%

Table 4: Detected threats in the captured darknet traffic in February 2012.

For further investigations in more detail, we analyzed four weeks (March 2012) of network captures and log files of the network services on a Linux Web server hosting the web page of our research group. Results showed that more than 69% of incidences are requests which resulted in "HTTP 404 Not Found", 17% are "SSHD authentication failure" and 14% assumed to all other incidences, for example "HTTP 400 Bad Requests" or failed SNMP requests.

2 http://www.snort.org

In order to have a more detailed view on these incidences, we deployed Linux honeypots providing various network services (sshd, httpd, sqld, ...) and we captured the network traffic for four weeks. Results showed that most of the network attacks are either brute force attacks or scanning attacks on the Web server in order to find default files corresponding to known vulnerabilities or to find popular misconfigurations (Table 5). Most used logins in the executed brute force attacks have been "root", "bin", "mysql" and "news" using either the same string as the password or "12345". On the Web server, attacks mostly searched for the files that can be seen in Table 6. We could identify more than 40 different files which are interesting for attackers since they have known vulnerabilities or they are indicative for misconfigurations. We considered these results in the triggering mechanisms of the dynamic honeypot service.

service	**protocol**	**occurrence**
httpd	tcp	19.35%
sshd	tcp	14.71%
MS terminal services	tcp	14.61%
remote administrator (radmin)	tcp	6.82%
telnet	tcp	4.32%
all others (e.g. standard sql services)	tcp	40.19%
WinMX File Sharing	udp	12.56%
Session Initiation Protocol (SIP)	udp	8.84%
Teredo tunneling	udp	3.72%
all others (mostly scans for P2P)	udp	74.88%

Table 5: Network services with most incoming connections on the deployed honeypots in June 2012.

/install/index.php
/install/setup.php
/install/configure.php
/mysql/translators.html
/phpMyAdmin/translators.html
/manager/html/index.html

Table 6: Most searched default files on web servers of the honeypots.

If an attacker identifies a vulnerable script on a web server, the script can already be exploited and cause errors. According to the results from our captured network traffic analysis, the honeypot service should definitely focus on preventing attacks against Web servers, especially prevent successful scans for vulnerable scripts or misconfigurations. Furthermore, the service should also prevent the successful execution of brute force or dictionary attacks, for example against

the SSH service. In particular, the dynamic honeypot service should protect from the two different network attacks:

- It should prevent scanning attacks on the Web server which try to identify default files, vulnerabilities or misconfigurations, but without risking to reveal sensitive information or the successful execution of attacks on the original VM.
- It should prevent brute force or dictionary attacks while not causing suspicion of the attacking source and not revealing any sensitive information.

5.1.2.2 *Architecture of the Low-interactive Honeypot Service*

In the following, we describe the design, the implementation and the evaluation of the proposed low-interactive honeypot security service. The service identifies attacks before they are successfully executed and it immediately deploys a new extracted honeypot VM that protects the original target VM. The service operates efficiently, economically and saves cloud resources by only using them once they are required. The major goal is to learn from executed attacks or scans and identify misconfigurations. The procedure of the low-interactive honeypot service can be described in seven sequential steps managed by a honeypot controller (Figure 26).

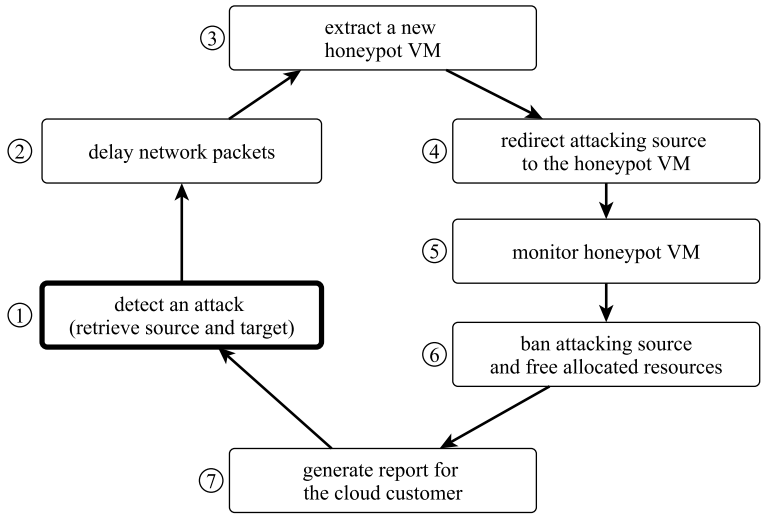

Figure 26: Flow graph of the proposed low-interactive honeypot service.

In step 1, the honeypot controller identifies an attack which targets at a customer's VM located on the same physical hardware. The identification of an attack is the trigger for the honeypot extraction and deployment procedure. First, the controller retrieves the IP address of the customer's VM which is the target of the ongoing attack and the IP address of the attacking source. In step 2, the controller delays the

attack until a new honeypot VM is extracted and was successfully deployed in step 3. The delay is very important since the ongoing attack should not be interrupted or disturbed. In order to have only a short delay, the extraction of the new honeypot VM needs to be performed in the range of seconds. In step 4, the controller redirects the traffic of the attacking source to the new deployed honeypot VM. In step 5, information about the ongoing attack is passively collected through the hypervisor layer. After a predefined period of time or after detecting the attack being successful, the honeypot VM is terminated and the attacking source is banned from the network in step 6. Finally, in step 7, a report for the cloud customer owning the original VM is generated. The report reveals potential vulnerabilities and misconfigurations. In order to prevent overloading or denial of service attacks, only one honeypot VM can be deployed on a physical hardware component at the same time.

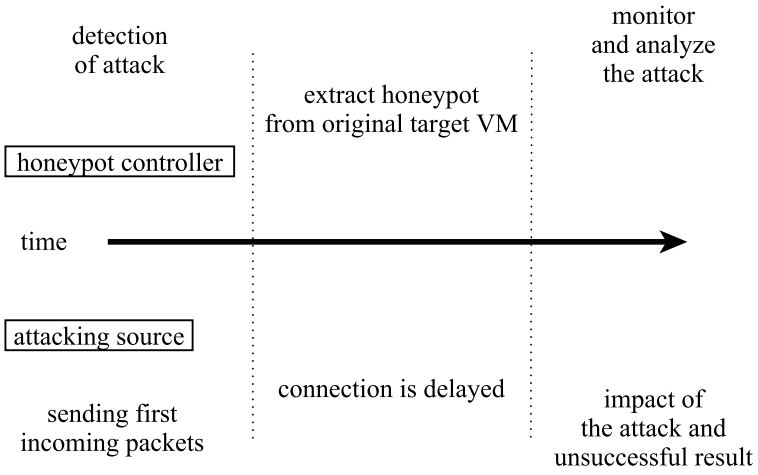

Figure 27: Time schedule of the honeypot controller.

Figure 27 illustrates the time line of the proposed security service. The attacking source sends the first network packets. The controller can assign these packets to a web directory scan, a brute force attack or it can identify a payload. The controller detects the attack and delays these packets and any further packets until a new honeypot VM is extracted from the original target VM. After the deployment, the packets are redirected to the new honeypot VM and the attack procedure can continue. From this point in time, the attack is precisely monitored and analyzed. The honeypot extraction process needs to achieve four different objectives:

1. The deployment of the honeypot VM needs to be fast because the detected attack should only be delayed for a short period of time. The architecture should not arise suspicion of the attacker and not interrupt the work-flow of any attack procedures, for example executed by automated tools.

2. Instead of having a cloned honeypot VM storing same data, we want to have a reduced honeypot VM without the real data. To reach this goal, the service removes previously defined data.

3. Once the controller detects an attack being successful on the honeypot VM, it immediately terminates the honeypot VM without revealing any information. Accordingly, continuous and precise monitoring of the deployed honeypot VM is required.

4. There should be no need to install or configure software on the original VM. All proposed mechanisms should operate transparently according to the SECaaS business model (Section 1.5).

DETECTION AND DELAY OF AN ATTACK: We used a Xen hypervisor setup (Section 2.2) to implement the proposed service. The controller operates on the administrative VM (dom0). It monitors the network traffic of all other VMs of the cloud customers hosted on the same physical hardware. The controller can detect brute force and dictionary attacks or Web server scans as well as payloads of certain known exploits. For this, we implemented a kernel module for the administrative VM using the Netfilter[3] framework. We defined and implemented three simple events which reveal the initiation or the first packets of attacks and which trigger the extraction of a new honeypot VM:

- **Repeats**: More than n new connections from the same source IP address to the same port in a fixed period of time (brute force or dictionary attacks).

- **Incoming data**: The content c_{in} was found in an incoming network packet, for example a bunch of No-Operation-Instructions (NOPs) indicating the shellcode of an exploit.

- **Outgoing data**: The content c_{out} was found i times in subsequent outgoing network packets targeting the same destination, for example simple HTTP 404 replies. Multiple 404 replies reveal a scan for default files.

Based on the results of our captured network traffic investigations in Section 5.1.2.1, we suggest using the values 5 for the parameter n, "\x90\x90\x90\x90" for c_{in} and "HTTP1.1 404 Not Found" for c_{out} with $i = 3$. More sophisticated rules can be defined as well as an advanced intrusion detection system can be used to trigger the extraction procedure. However, we demonstrate the feasibility of our approach using these three simple event classes. The controller can detect attacks before they are forwarded on the virtualized network device or during the attacks are performed in the early stages.

3 http://www.netfilter.org/

It buffers the network packets and identifies packets which break a defined rule. Once a new honeypot VM is deployed, the malicious connection is redirected.

EXTRACTION OF A HONEYPOT VM: Since booting a new honeypot VM takes too much time, the honeypot VM is live cloned from a running VM. We modified the source code of the Xen hypervisor in order to enable live cloning. In particular, live migration is already supported by the Xen hypervisor [18]. This operation enables the transfer of a running VM from a physical hardware component to another physical hardware component only with minimal interrupt (Section 2.3). We changed 180 source lines of code to enable live cloning on the same physical hardware. Live cloning uses the iterative copy procedures of live migration, but does not delete the source VM as well as it maintains emulated hardware for both VMs.

Live migration copies the volatile memory, but it does not copy the storage of the VM, which is usually located on a network file system. However, creating a useful replica of a VM also includes copying the storage. Since the storage of a VM can be very large with many GB of data, a copy process would take too much time for our purposes. Our goal is to extract and deploy a new honeypot in the range of seconds.

In order to reach this goal, we use the Linux logical volume manager (lvm2) and create a "copy-on-write" virtual snapshot of the VM's storage during run-time. The virtual snapshot can be mounted on the cloned VM even with read and write access. Before the cloned VM can be used as a honeypot and the attacking source is redirected, several adjustments need to be performed on the cloned VM. First, the honeypot controller mounts and modifies the snapshot before assigning it to the cloned VM. It replaces the Linux shadow file with a dummy shadow file in order to avoid the loss of real passwords through successfully executed brute force or dictionary attacks. From now on, it is not longer possible to log into the cloned VM by using network services. This way, the controller can allow the continuation of brute force or dictionary attacks which will never be successful on the honeypot VM. Furthermore, the controller removes sensitive files previously defined by the cloud customer. The controller deletes files in the web directory which causes HTTP 404 replies once the scans continue. However, the path and name of each deleted file is stored in a list for later analysis and report generation. Finally, the extraction procedure created a modified running cloned VM which we use as a new honeypot VM.

Since the new honeypot VM maintains the same IP address as the original VM, we need route the network packets of the attacking source depending on their source IP address to the honeypot VM. This way, we create the illusion for the attacker to communicate with the original VM instead of the honeypot VM. Furthermore, outgo-

ing traffic from the honeypot VM to an IP address which does not belong to the identified attacking source and outgoing traffic from the original VM targeting the attacking source needs to be blocked. Other connections which belong to the normal work-flow and which are not identified as malicious traffic are routed to the customer's original VM without any interruptions.

MONITORING OF THE HONEYPOT VM: Before enabling the dynamic honeypot service, the cloud customer needs to define certain information in a configuration file. The configuration file defines the directories containing the web content (for example */var/www/*) and which log files need to be monitored. The controller uses three different techniques to monitor the honeypot VM. First, the honeypot controller uses VMI (Section 2.4) in order to transparently monitor the volatile memory of the honeypot VM. The controller monitors the list of running processes and loaded modules on the honeypot VM including their corresponding PID. The controller immediately detects modifications, for example if a new process is started or a new module is loaded. Changes in these lists lead to the immediate termination of the honeypot VM. Second, the controller uses live forensic techniques (Section 2.5) to examine the honeypot VM's raw storage (the virtual snapshot). The honeypot controller transparently extracts relevant information about ongoing events on the honeypot VM from known log files. This way, the controller investigates, for example which usernames have been tried in brute force or dictionary attacks as well as for which files the web server has been requested. Information about the requested files is later used for comparison with the list of deleted (but previously existing) files on the original VM. Third, the controller uses the virtualized network device to monitor the incoming and outgoing network traffic. After a defined period of time, the honeypot VM is terminated and the allocated resources are freed.

REPORT GENERATION: After termination of the honeypot VM, the controller generates a report for the cloud customer. The report includes information about performed requests for files on the web server. In particular, this information includes the requests which would have been successful and would actually find a target file on the original VM based on the comparison of the list of existing files before deleting them from the web directory on the honeypot VM. Furthermore, it includes information about the login attempts and details about other incidences, for example, if a process was terminated. In the following, an example report can be seen. The attacker scanned for misconfigurations on the Web server and would have found a default setup script which the customer may forgot to delete. Furthermore, the attacker performs a dictionary attack against the

SSH service. Most interestingly, the attacker sends data to the IMAP port (143) and right after that a new bash shell process was started which lead to the immediate termination of the honeypot VM.

```
<attack id="0701">
     <alert>1</alert>
     <timestamp>06/21/2012 00:37:00</timestamp>
         <event>
            <tcp>80</tcp>
            <found>/cms/assets/setup.php</found>
         </event>
         <event>
            <tcp>22</tcp>
            <failed-login>root</failed-login>
            <failed-login>mysql</failed-login>
         </event>
         <event>
            <tcp>143</tcp>
            <process>/bin/bash</process>
         </event>
     <info>Change file permissions on the web server
     and immediately update the IMAP daemon.</info>
</attack>
```

With the help of the generated report, the customer can reveal misconfigurations and vulnerabilities which an attacker could successfully exploit on the original VM. This is based on the fact that the extracted honeypot VM runs exactly the same software in the same configuration.

5.1.2.3 *Evaluation of the Low-interactive Honeypot Service*

For the evaluation of the proposed architecture, we used a modified Xen hypervisor 4.1 and a server with an Intel(R) Xeon(R) Quad CPU X3450 with 2.67 GHz and 4 GB of memory. The administrative VM (dom0) runs Ubuntu Linux 10.04 (i386) and the customer VM runs Ubuntu Linux 11.10 (i386). In our setup, creating a new lvm2 snapshot of the original VM's storage which is assigned to the honeypot VM is very fast (200 ± 45 ms) and does not depend on the size of the original storage. Mounting the snapshot and replacing or deleting certain files (shadow file, file in the Web directory) can be performed in the range of milliseconds as well. The time needed for the entire honeypot extraction procedure is mostly influenced by the time required for the live cloning process which iteratively copies the volatile memory of the original VM in order to create a consistent replica. Results can be seen in Table 7. A honeypot VM cloned from an original VM with 2 GB of memory can be live extracted, modified and deployed in less than 7 seconds. This makes the proposed service feasible in realistic scenarios since network attacks like dictionary attacks or scanning for default files on a Web server usually need to be executed over quite longer periods of time in order to be successful.

Due to the underlying live migration algorithm, the more memory pages are modified during a copy procedure, the more copy iterations

VM memory size	512 MB	1024 MB	2048 MB
timing	2.79 ± 0.17s	4.63 ± 0.29s	6.92 ± 0.64s

Table 7: Low-interactive honeypot VM extraction procedure for different memory sizes. Average of 32 runs.

have to be performed. If the original VM is highly utilized, the live cloning procedure can take more time. In order to improve this, we modified the live migration algorithm and limit the number of performed copy iterations to three. This results in a constantly fast live cloning procedure even if the original VM of the cloud customer is highly utilized.

In this section, we investigated how we can built a flexible *low-interactive* honeypot service for cloud customers. In the next section, we we design, implement and evaluate a *high-interactive* honeypot service. In this context, triggering the system once an attacker has already access to an original VM, removing sensitive data and continuously maintaining the illusion of the attacker to be still logged in the original VM are of particular interest.

5.1.3 *High-interactive Honeypots for Attack Analysis*

In this section, we present a security service that extracts a high-interactive honeypot from the VM of a cloud customer. The extraction procedure is triggered once an attacker has already gained access to the operating system. Once an intruder could be identified, the intruder is extracted from the VM by cloning the VM and transforming the cloned VM into a high-interactive honeypot VM running co-residently on the same physical hardware. During this cloning procedure, sensitive data is removed or blocked from the VM's memory as well as from the VM's storage. Thus, the new honeypot VM is a replica which runs the same software in the same configuration but without sensitive data. Most importantly, the established connection of the intruder is not terminated, but is maintained and redirected. Additionally, certain modifications are performed on the honeypot VM in order to maintain the intruder's illusion to be still located in the original VM. Finally, the service transparently monitors the intruder's actions and intentions. More precisely, we make the following contributions:

- We developed a VM live cloning procedure which removes critical data and inserts essential administrative data "on-the-fly". An identified intruder is extracted into the cloned VM while other logged-in users remain in the original VM. All established remote connections are maintained, no connection is interrupted.
- We implemented several techniques to transform the cloned VM into a unique high-interactive honeypot VM by executing specific modifications depending on properties of the original VM's data. This way, we maintain the intruder's illusion to be still connected with the original VM.
- We monitor the new honeypot VM in order to understand the intruder's actions and intentions. The monitoring procedures only use transparent techniques. There is no need to install additional software on a customer's VM which is protected by the high-interactive honeypot service.

In the work of this section, we do not deal with detecting intrusions since this problem is complementary to our work and several existing technologies for intrusion detection based on virtualization technologies can be found ([29], [64]). Rather, we focus on the honeypot extraction, modification and monitoring procedures as well as on the network management.

5.1.3.1 *Intrusion Analysis*

In this attack scenario, we assume an intruder who has already gained access to a customer's VM and who tries to escalate obtained privi-

leges or investigates certain data on the compromised VM. In general, gaining access to a remote operating system is often reached with the help of performing two major attacks: Guessing or stealing valid login credentials or remotely exploiting software vulnerabilities on the target operating system. Brute force attacks targeting network services like SSH are the most common attacks in computer networks. Although there are a lot of approaches for fast detection [73], novel prevention methods [82] or password generation methods [25], the utilization of secure login credentials still remains an open problem. In a study based on deployed honeypots, Jim Owens et al. [60] had more than 400 brute force attacks on each machine on every single day. Often, valid login credentials are even stolen directly from the client with the help of phishing attacks [20]. Furthermore, a system can also be compromised without the need for valid login credentials, but by remotely exploiting software vulnerabilities with the help of buffer-overflows, format string vulnerabilities and broken or risky implemented authentication algorithms [23].

5.1.3.2 *Architecture of the High-interactive Honeypot Service*

The high-interactive honeypot service operates on the administrative VM (dom0) located co-residently with other customer VM's on a pyhsical hardware component. The different procedures which are sequentially executed by the service are illustrated in Figure 28.

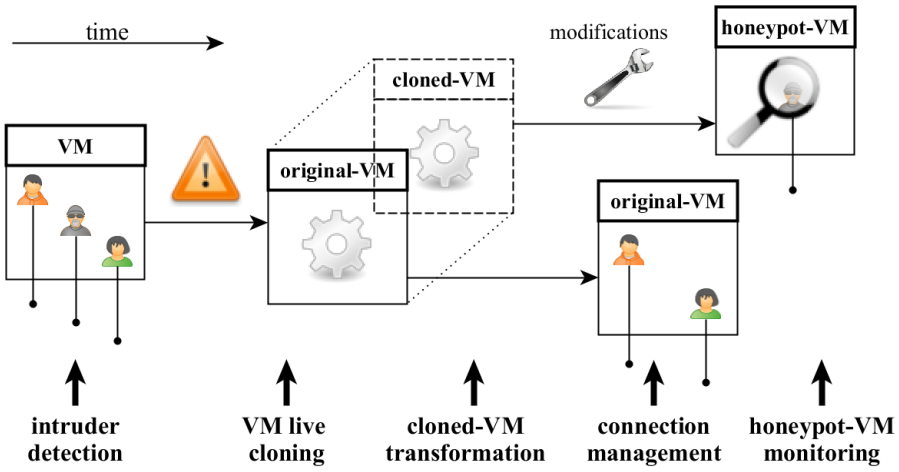

Figure 28: Procedures of the high-interactive honeypot service isolating and monitoring an intruder on-the-fly.

First, an intruder is detected in a customer VM and distinguished from other users who can work on the same VM in parallel. The detection of an intruder immediately triggers the VM live cloning procedure which finally results in a replica of the VM. During and after the cloning procedure, the cloned VM is rapidly modified and transformed into a high-interactive honeypot VM. Established connections are maintained, the connection of the intruder is redirected to the hon-

eypot VM without interruption and without influencing other users on the original VM. Finally, the honeypot VM is precisely monitored in order to obtain information about the intruder's actions, intention and potential vulnerabilities. After a defined lifetime, the honeypot VM is destroyed, the intruder is banned from the network and a report with specific information about the incident is generated and sent to the customer.

INTRUDER DETECTION: In order to trigger the honeypot extraction procedure, an intruder has to be detected and distinguished from all other users who do not have malicious intentions but who can work on the same VM in parallel. The service periodically extracts the authentication log file from the original VM in order to continuously match the IP addresses of established connection to the logged-in users. For intrusion detection, the service monitors four different points on a Xen hypervisor setup (Section 2.2) and continuously retrieves real-time information about operations on the VM. VMI (Section 2.4) is used to gather information about the running processes and loaded modules. Live storage forensics are used to examine the raw storage of the VM by extracting meta information of previously defined files to detect modifications. Furthermore, the service retrieves information about the network traffic from the virtualized network device as well as information about the VM's resource utilization from the hypervisor. The collected information can be used to feed an arbitrary intrusion detection system to detect an intruder and to trigger the honeypot extraction procedure.

The high-interactive honeypot extraction procedure consists of two different phases: First, there is a cloning phase which creates a modified cloned VM from the original VM similar to the live cloning procedure proposed in Section 5.1.2 in which we created a low-interactive honeypot VM. The cloned VM runs the same software in the same configuration but without sensitive data. Second, there is an adaptation phase which finally transforms the cloned VM into a unique high-interactive honeypot VM.

VM LIVE CLONING: Once the extraction process is triggered, the VM is live cloned during run-time. In order to reach this goal, we modified the Xen live migration algorithm [18] to support live cloning of a VM on the same physical hardware component (Section 2.3). Figure 29 depicts the phases of the live cloning approach. At the beginning, the same iterative live migration algorithm is used, but slightly modified to enable live cloning and the utilization of optimized parameters.

The honeypot service performs an additional last iteration in the cloning procedure in order to perform certain modifications in the volatile memory. The VM's volatile memory may contain sensitive

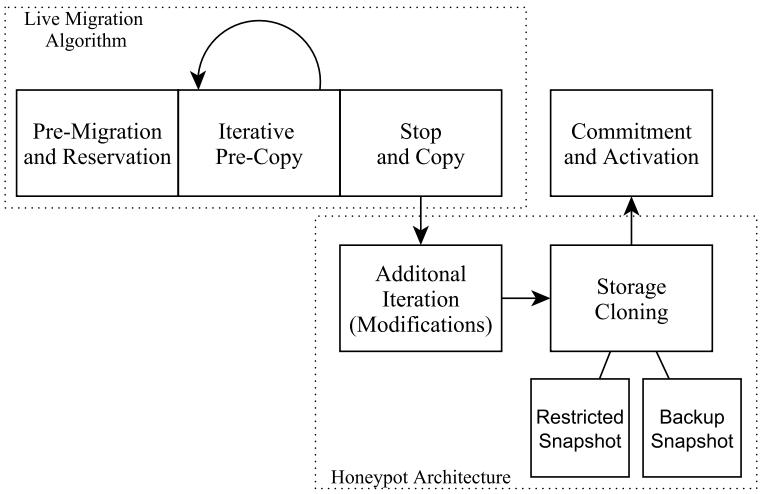

Figure 29: The extensions added to the Xen live migration algorithm in order to enable live cloning with modifications to create a high-interactive honeypot VM.

data which needs to be removed. For example, if the Linux "sudo" shell command was executed, it temporarily stores the plain root password in the volatile memory. Furthermore, the Linux PAM modules (Pluggable Authentication Modules) store the Linux password hashes of the users in the volatile memory. Sensitive data, even if it is only a password hash, needs to be removed in the cloned VM's memory. An attacker who is logged-in the high-interactive honeypot VM could retrieve the real password hashes and execute dictionary attacks in order to reveal valid passwords. The service removes the sensitive data with the help of regular expressions of previously defined known patterns. A pattern $P_{c,l}$ is a fingerprint of initial unique characters c and the length l of the target sensitive data. The service can remove arbitrary sensitive data in the volatile memory based on multiple patterns $P_{c,l}$ which can be defined in a configuration file. Technically, removing data from the VM's volatile memory means replacing the located sensitive data with dummy data having the same length and characteristics. Additionally, the service can also insert specifically prepared data into defined locations. In particular, it replaces the root password hash in the cloned VM's memory with a previously defined hash during the modifying cloning process. The replaced password hash allows the honeypot service to access the cloned VM with a valid account after the procedure is finished.

In order to create a fully cloned VM, the service needs to add a copy of the original VM's storage to the replica. However, the storage of the customer's VM can be very large. Again, the service uses lvm2 snapshots like we used in the low-interactive honeypot architecture described in Section 5.1.3, but this time it creates two snapshots of the VM's storage before the cloned VM is activated. The first snapshot is not used but required for later comparison in order to rapidly

reveal new or deleted files on the honeypot VM. The second snapshot is first modified and then added to the cloned VM. Files which are not system-relevant or arbitrary files which are sensitive are blocked by the service. To reach this goal, the service creates a linear lvm2 snapshot which matches the cloned VM's storage sectors to the original VM's storage sectors but it blocks specific data by intentionally setting IO errors on the corresponding sectors. This way, arbitrary sensitive data can not be accessed on the snapshot after it is added to the cloned VM. Since the storage can be very large, blocking the sectors of sensitive files instead of deleting the files can be performed much faster. Furthermore, the blocked files cannot be restored and the operation cannot be undone from within the cloned VM. In the standard configuration, the service blocks all files in the */home/*, */tmp/* and the */root/* directory. Nevertheless, arbitrary directory paths can be defined by the customer. The sectors on the storage on which the sensitive files are located, have to be known to the service before the cloning process is triggered. Starting to locate the sectors of the sensitive files during extraction would take too much time and dramatically slow down the procedure. As a solution, the service continuously monitors the sectors of the given sensitive files by periodically examining the original VM's raw storage with transparent storage forensic techniques (Section 2.5). This way, it always maintains an up-to-date list of files and their corresponding sectors on the storage. Once an intruder is detected and the honeypot extraction procedure is triggered, the sectors of the sensitive files which have to be blocked during the cloning process are already known. After sensitive files have been blocked, their names and absolute paths are stored by the service in a list L. Later, the list is used for custom-made adaptation and transformation of the cloned VM into a unique high-interactive honeypot VM.

Finally, the live cloning procedure results in a cloned VM without sensitive data in the volatile memory, without the opportunity to access sensitive files on the storage and with an additionally inserted root account that gives access to the service. However, the original VM and the cloned VM maintain the same MAC address and the same IP address. Fine-grained network management needs to be performed in order to distinguish the logged-in users from the logged-in intruder, but first the cloned VM needs to be transformed into a high-interactive honeypot VM.

TRANSFORMATION OF THE CLONE: The service transforms the cloned VM into a high-interactive honeypot VM which should be able to reveal the actions and the intentions of the intruder, but should not arouse the intruder's suspicion to be fooled. The final step in the transformation of the cloned VM requires several adaptations specifi-

cally made to maintain the intruder's illusion to be still located in the original VM. For the adaptations, the inserted root account is used.

Until now, the service modified the cloned VM's volatile memory as well as it blocked sectors of sensitive files on its storage. However, on the storage, the original password hashes can still be found in the Linux shadow file and refreshing the Linux page cache would simply replace the modifications in the volatile memory with the original data. In order to avoid this, the service replaces the file with a new shadow file defining dummy password hashes for accounts and the same root password hash previously used for account insertion in the volatile memory. Furthermore, the service also terminates and removes running processes which are unwanted on the honeypot VM (for example backup processes). The names of the processes can be defined in the configuration file. Finally, the service drops the cloned VM's memory cache. At this point in time, every critical and sensitive data is either completely removed or blocked.

For further adaptations, the service modifies three popular shell tools which belong to the GNU Coreutils[4] and the GNU Findutils[5]. These modifications depend on the list of blocked files L created by the service during the cloning process. The service creates new versions of the *ls*, *cat* and *find* tools especially for the honeypot VM. The modifications maintain the illusion that the blocked files of L still exist and they prevent the intruder receiving IO errors when using these commands on the blocked files.

The modifications of these three commands help to get information about the intentions of the intruder and to maintain the intruder's illusion to be still located in the original VM. The service can reveal if the intruder had a primary target, for example if the intruder tried to find specific files. The service creates a second list L_r containing a new random name for each file stored in L generated with an English dictionary. In detail, the performed modifications on the shell tools are as follows:

- **ls**: The service modifies the source code of the directory listing command *ls* which finally does not display the names of the files contained in L if requested, but instead displays the corresponding new file name stored in L_r in the same directory. This way, sensitive information which can be contained in the real file names are removed. However, the hierarchy of the real file system is maintained and used to monitor if the intruder searches for specific information.

- **cat**: Since the modified *ls* shell command displays new file names for the removed files in L, but the real data of these files cannot be accessed, the service also modifies the sources of the *cat* com-

4 http://www.gnu.org/software/coreutils/coreutils.html
5 http://www.gnu.org/software/findutils/findutils.html

mand which displays newly generated content once a file of L_r is accessed which actually does not exist but can be listed with the modified *ls* command. Once the modified *cat* shell command is executed, it immediately downloads, filters and displays the text content of a randomly selected Wikipedia article.

- **find**: Searching for and finding files which do not exist is enabled by modifications in the *find* shell command. A modified version of the *find* command actually creates the files for which the intruder searches in a randomly selected directory.

The unique modified sources depend on L and L_r and they are newly compiled each time a new honeypot VM is created and they are inserted before the Linux page cache is refreshed. Of course, more shell commands can be modified to provoke actions of the intruder as well as shell commands which the intruder should not execute can be removed.

Finally, if all steps are successfully executed, the transformation of the cloned VM into a custom-made high-interactive honeypot VM is finished. The established connection of the intruder can now be redirected by the service to the new honeypot VM.

CONNECTION MANAGEMENT: Connection management is a very important step in establishing the new high-interactive honeypot VM. The intruder who already has gained access to the original VM maintains an established remote connection which should not be interrupted. An interruption of the connection would raise the intruder's suspicion that something went wrong. Furthermore, the network activity of other normally logged-in users on the original VM should not be affected by our procedures. Since both systems, the original VM and the new honeypot VM, maintain the same IP address, this step is not easy to implement.

We designed the following networking setup: During the cloning and transformation process, all traffic from and to the cloned VM is blocked. Once the honeypot VM is finally created, traffic is diverted based on the source IP addresses. Traffic from the intruder's IP address is rerouted to the honeypot VM, while the remaining traffic (for example the traffic of other logged-in users who work on the original VM in parallel) is still routed to the original VM. Since there are open network connections of the intruder on the original VM as well as open connections of other users on the honeypot VM, the outgoing traffic of both VMs needs to be precisely managed. If any errors occur, packets of connection timeouts could interfere and interrupt established connections. Outgoing traffic from the original VM to the intruder is dropped and outgoing traffic from the honeypot VM to other users is dropped as well. Figure 30 shows two normal users communicating with the original VM while the detected intruder is

routed to the honeypot VM. In both cases, outgoing traffic of the other VM is blocked to prevent connection errors.

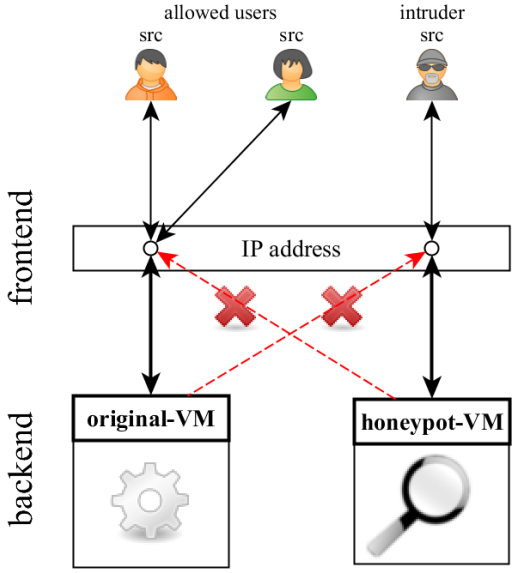

Figure 30: Connection management and redirection of the intruder to the honeypot VM while not affecting others on the original VM.

It might be required that the intruder can also communicate with other hosts on the Internet, for example to download additional exploits from a third party server, or that malware can communicate with a command and control server. For such setups, we decided to use the following strategy: The honeypot VM is allowed to communicate with other destinations besides the intruder. Only certain hosts internally used by the cloud as database servers, replicated storage or backup servers are excluded, because such communication could cause data inconsistency and disrupt the operation of the original VM. However, existing outgoing connections, being established before the cloning process are not allowed and corresponding packets are dropped. For new connections, the masquerading feature of the Linux networking stack is used and these connections are mapped to a new IP address. Without masquerading, both VMs might choose the same source and destination port numbers to communicate with a remote server, which could not be correctly handled. These properties are implemented on the virtualized network device of the Xen hypervisor setup. In addition to these static rules, dynamic limits as the rate of new SMTP connections preventing the intruder starting to send spam emails or the total bandwidth preventing denial of service attacks can be also deployed.

Finally, after the high-interactive honeypot VM is created and the connection management is executed, the intruder is already extracted from the original VM since the intruder's presence is now isolated

and located on the honeypot VM. From the intruder's perspective, this procedure is hardly noticeable because there is no connection interruption nor any other obvious signs on the system. The intruder is sill connected with a VM running the same software in the same configuration.

HONEYPOT MONITORING: Once the honeypot VM is successfully created and the intruder is redirected, the honeypot VM is transparently monitored by the service to reveal the intruder's course of actions. Basically, it uses the same transparent monitoring techniques which are also used in Section 5.1.2, but it collects more details. The services used VMI as well as live storage forensics, information from the Xen hypervisor and from the virtualized network devices.

The honeypot VM is only deployed for a previously defined lifetime in which it is precisely monitored. The service continuously retrieves and stores the following information as long as the honeypot VM is deployed:

- **System**: The service periodically retrieves a timestamped list of running processes by accessing the honeypots VM's memory via VMI (Section 2.4). It also gathers their *uid*, *pid* and used arguments as well as a timestamped list of loaded kernel modules by walking the kernel's module list. New started or terminated processes or modules are identified in chronological order. Furthermore, the service transparently retrieves a list of currently open network sockets and logs the Linux message buffer (dmesg).

- **Files**: The service continuously extracts certain configuration files by examining the raw storage with live storage forensic techniques (Section 2.5). The service periodically computes a hash over the data of these files and this way it can immediately reveal modifications in chronological order. Furthermore, it continuously extracts and stores the shell history files for later analysis of the entered shell commands. If the intruder cleans the log files, the information is already stored outside the honeypot VM.

- **Communication**: The service records all data sent and received by the honeypot VM over the network including protocols, ports and destination IP addresses.

ANALYSIS AND REPORTING: The general idea of the proposed high-interactive honeypot service is to analyze what kind of actions an intruder has performed and what intentions the intruder may had. In order to support the original VM's customer in investigating the incident, the service generates a report based on the analysis of the collected data. The report contains relevant information about the in-

cident and the modifications on the honeypot VM in detail and in chronological order.

Once the lifetime of the honeypot VM is exceeded, the service destroys the honeypot VM and bans the intruder from the cloud network. In order to reveal new files or deleted files on the honeypot VM, the service compares the snapshot of the storage which was assigned to the honeypot VM with the second backup snapshot also created during the cloning procedure. This way, new, deleted and modified files can be rapidly revealed.

As an additional feature which can be used for investigation, the service tries to identify the intention of the intruder based on the collected information. For this, we defined three different intention classes and the service classifies the intruder to one intention depending on the recorded activity:

- **Investigation**: The intruder searched for specific information on the compromised system. Further investigations can also reveal if the intruder already had previous insider information about the system or an institution.
- **Misusage**: The intruder manipulated the system for its own purposes to perform further malicious activities. For example, the intruder deployed software which is used for further attacks against other systems.
- **Malfunction**: The intruder tried to destroy the system or to intentionally collapse the system by causing malfunctions.

In order to classify the intention of an intruder, the service creates a feature vector V from the collected information. V consists of the sub-vectors $\{V_s, V_f, V_t, V_c\}$. V_s contains the number of new started unknown processes and new loaded modules. V_f contains the number of new created files, the number of modified files as well as the number of deleted files. V_t contains the amount of incoming traffic to the honeypot VM in bytes as wells as the amount of outgoing traffic and the number of contacted remote source IP addresses. Finally, V_c contains the number of the executions of the *ls*, *cd* and *cat* shell commands and the number of the file name which was entered the most. The service generates V and classifies this vector to one of the intention classes by using a heuristic approach based on statically implemented rules. We defined simple classification rules based on a number of performed experiments. The resulting probabilities are added to the report and can help to decide if further manual investigations are required or they can give a quick overview to the cloud customer.

The service generates a report for the VM's customer based on the recorded and analyzed data collected on the honeypot VM. The report reveals potential risks and vulnerabilities as well as which software patches should be installed. Table 8 shows an example report

after analysis based on an incident triggered by an intruder who obtained valid login credentials for the "foobar" account.

honeypot report	#3
intruder detected	23:54 - 5/7/2013 accessed honeytoken
honeypot lifetime	00:04 - 6/7/2013
compromised account	*foobar*
source IP address	*(anonymized)*
generated traffic	130718 bytes
dst IP address contacts	*(anonymized)* *(anonymized)*
started processes (4)	 buf -offset 128 buf -offset 512 turtle -hide -amd64 obfuscate -s ubuntu -a foobar
new files (2)	 */tmp/expl.tar.gz* */tmp/tools.tar.gz*
modified files (3)	 */etc/shadow* */var/log/wtmp* */var/log/auth.log*
intruder intentions	 5% investigation 80% misusage 15% maloperation

Table 8: Example report after an intruder was monitored, the collected data was analyzed and the honeypot VM was destroyed.

The report shows the point in time the intruder was detected and the mechanism used for detection ("honeytoken"). To get an insight into what actually happened on the honeypot VM during its lifetime, the report shows unknown started processes, newly created files and modified files. Finally, based on the recorded data, the probability for each of the three intention classes is added to the report.

5.1.3.3 *Evaluation of the High-interactive Honeypot Service*

The high-interactive honeypot security service is implemented for Linux for a Xen hypervisor setup 4.1 (Section 2.2) operating in routing mode. We modified 248 lines of code on the Xen hypervisor code in order to implement the modifying live cloning procedure. The service is programmed in Python with 1307 lines of code. The service uses fine-grained VMI (Section 2.4) and a modified version of the Volatility[6] forensic framework which has been extended by specific plugins

6 http://code.google.com/p/volatility/

tailored towards our purposes. Live storage forensic techniques are used in order to transparently examine the raw storage during runtime (Section 2.5). Customer VMs use an ext3 file system and lvm2 volumes for storage which allows the service to create live snapshots.

For an evaluation, we used an Intel(R) Core(TM) 2 Duo CPU with 3 Ghz and 4 GB of memory. We investigated the honeypot extraction procedure and the different transparent monitoring techniques.

CLONING AND TRANSFORMATION: We focused on the honeypot extraction procedures and only implemented a simple but effective intruder detection approach. If an attacker has gained access to the VM, for example based on a successful dictionary attack, the service detects the intruder as follows: We create executable files which have interesting names but actually no effect on the customer VM's file system. We call these files "honeytokens". A honeytoken can attract a curious intruder and detection of the intruder accessing the honeytoken reveals her and triggers the honeypot extraction procedure. This approach is quite simple, but effective, since we can store a greater number of honeytokens on different locations on the file system.

In order to determine the overhead of the proposed modifying live cloning process which removes sensitive data and inserts a root account on the cloned VM, we deploy a customer VM on which the password hashes of three different user accounts, the plain root password stored by the *sudo* shell tool and a private key (256 bit) needs to be located and removed from the volatile memory. Figure 31 shows normal live cloning processes without our modifications in comparison to the live cloning process performed by our service which uses an additional iteration in order to scan the volatile memory and replace sensitive data.

The measurements depend on the current utilization of the customer's VM. Figure 31 shows the cloning process during high memory utilization without memory modifications in comparison to a live cloning process during high memory utilization. Our modifications generate an average overhead of around 20%, of course increasing with increasing amount of the original VM's memory.

The service periodically retrieves the corresponding sectors of given sensitive files on the customer VM's raw storage (for example in */home/*, */var/lib/*, */tmp/* and */root/*). The service continuously monitors these files during normal operation even if there is no intruder in order to respond quickly if an intruder could be identified and sensitive files have to be blocked rapidly. Retrieving the sectors of files on the storage once an intruder was detected would dramatically slow down the honeypot creation procedure, because the number of sectors of the sensitive data can be very large. In our setup, the service retrieves the sectors of 50 different files which are all located in different directories in 3.1 ± 0.1 seconds (207294 single sectors). The service

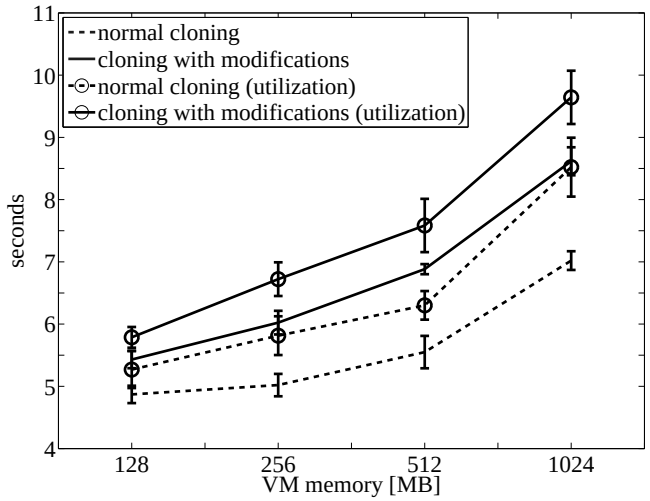

Figure 31: Duration of standard live cloning in comparison to live cloning with our proposed memory modifications (30 test-runs).

maintains an up-to-date list of the corresponding sectors. If a new cloning procedure is started, a list of the sectors is already available and they can be rapidly blocked on the new storage snapshot created for the cloned VM.

The creation of the custom-made shell tools require different modifications in their source code each time a new honeypot VM is created. Creating unique *coreutils* and *findutils* depending on the number of blocked files can be seen in Table 9. The new modified shell tools can be created and deployed in around 2.0 ± 0.1 seconds. The procedure can run in parallel to the cloning procedure.

description	**timing**
unique coreutils adaption and creation	0.988 ± 0.143 s
unique findutils adaption and creation	0.342 ± 0.071 s
insertion into cloned VM	0.642 ± 0.117 s
synchronizing the Linux page cache	0.107 ± 0.005 s

Table 9: Adaptation of shell tools for a new honeypot VM (30 test-runs).

Finally, redirecting the intruder to the new honeypot VM while maintaining the established connections of other users and deploying several specific firewall rules to block traffic from the original VM to the intruder as well as from the honeypot VM to the normal users can be rapidly achieved in 53.53 ± 2.17 ms. In our test-runs, the intruder was never able to notice the extraction procedure and his relocation to a new honeypot VM.

All in all, cloning the original VM with modifications, transforming the cloned VM into a custom-made unique honeypot VM by re-

placing shell tools and rerouting an intruder can be performed in approximately 7 seconds depending on the current workload of the customer's VM (512 MB memory). This period of time is little enough to prevent the intruder from deep investigations on the original VM.

HONEYPOT VM MONITORING: Once the new extracted honeypot VM was successfully deployed and the intruder is redirected, the honeypot VM is continuously monitored in order to record and analyze the intruder's actions. Table 10 shows timings of the periodic execution of different transparent monitoring techniques.

retrieved information	**required time**	**source**
processes and modules	1.55 ± 0.004s	VMI
bash history	0.23 ± 0.003s	storage forensics
system config files (7) and SHA1	0.83 ± 0.007s	storage forensics
system log files (6)	0.41 ± 0.005s	storage forensics
network information	0.17 ± 0.002s	hypervisor
resource utilization	0.14 ± 0.003s	hypervisor

Table 10: Timings of the periodically and transparently executed tasks in order to monitor the honeypot VM (30 test-runs).

Retrieving information about running processes, their corresponding *pid* and *uid*, and about loaded kernel modules takes approximately 1.5 seconds for more than 70 different processes and 20 loaded modules on the honeypot VM. Retrieving the *bash_history* file takes approximately 0.2 seconds, retrieving 7 essential system configuration files and computing and comparing their hash values to reveal modifications takes approximately 0.8 seconds. Retrieving information about 6 exemplary log files located on the honeypot VM's raw storage using the live forensic techniques takes approximately 0.4 seconds. Retrieving information from the virtualized network device about established connections and their destinations takes approximately 0.2 seconds and information from the hypervisor about current resource utilization approximately 0.1 seconds. Since several of these processes can run in parallel, all required information from a running honeypot VM which is used to analyze the intruders actions can be transparently and periodically retrieved in approximately 1.6 seconds. The resulting timings show the feasibility of our proposed high-interactive honeypot service which can beneficially be deployed by cloud customers in order to analyze attacks against their specific software setups.

In the first security service proposed in Chapter 5, we used virtualization technologies in order to create a replica of a customer VM which we finally transformed into a honeypot VM. Creating a separated environment with the help of a newly cloned VM has several advantages, like for example the availability of an entire isolated operating system. Assigning a separated VM to an attacker is quite resource-consuming, but very beneficial for analysis and easy to implement in an IaaS cloud (Section 1.4). However, this strategy fits not well in every scenario.

In this section, we briefly show a light-weight strategy for attack analysis also based on software virtualization [4]. Our approach is especially interesting for an "Software-as-a-Service" (SaaS) business model, in which single applications are hosted in the cloud and provided to customers, but it can be also used in an IaaS cloud. In this context, we do not use an entire hypervisor setup, but we use software virtualization on the application layer with the help of Linux Containers (LXC[7]) and the Checkpoint and Restore In Userspace project (CRIU[8]). LXC runs arbitrary Linux processes in virtual containers within the userspace of a Linux operating system. The CRIU project enables checkpoint and restore functionality for processes, which allows to create a synchronous dump of a target running process. Subsequently, we can restore the dumped process in a separated environment realized as a virtual LXC container. Figure 32 illustrates the architecture which clones a process into a separated LXC environment triggered and managed by a remote controller which additionally functions as a reverse proxy. Using this approach, we can trap a detected attacker whose connection is currently handled by a process and analyze the further course of actions.

The proposed architecture is quite similar to the proposed honeypot security services presented in Section 5.1 which are based on a Xen hypervisor setup. However, this time we only clone a single process within the same userspace of an operating system. This brings the advantage that the procedure can be performed faster while it also requires less resources. In the first step of Figure 32, the attacker is identified. In step two, the cloning procedure is triggered and executed within the same userspace. In step three, the attacker is rerouted to the new cloned process managed by a reverse proxy while already established connections are not interrupted.

In our exemplary scenario, the process is an Apache2 web server and an attacker tries to exploit a known software vulnerability which has already been patched. In particular, the service is triggered by a manipulated security patch which we called "honey-patch". A honey-

7 https://linuxcontainers.org/
8 https://criu.org

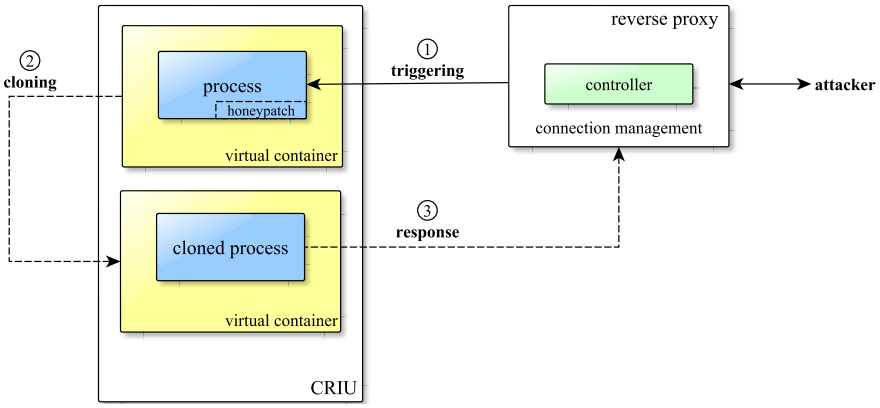

Figure 32: Architecture of the proposed fine-grained attacker trapping mechanism based on light-weight software virtualization.

patch closes a software vulnerability, like traditional security patches do, however, it additionally triggers the cloning procedure and the reverse proxy to redirected the attacker to a new replica which then allows the exploitation. Subsequently, the cloned Apache2 process is used as an isolated honeypot to investigate the attacker's further capabilities, strategies and intentions.

5.2.1 *Fine-Grained Memory Redaction*

An attacker who successfully exploited a vulnerability can eventually retrieve sensitive information, for example about sessions of other clients stored in the volatile memory of the cloned process. In the exemplary Apache2 web server scenario, sensitive information are for example the IP address or the request history of other clients. Especially information about encrypted connections is sensitive, since it can be used for other attacks, for example for Man-in-the-Middle attacks.

To solve this issue, we introduced a memory redaction procedure that replaces client session data with forged anonymous data.

Apache2 stores client sessions in well-defined data structures, for example the *request_rec* struct variable stores information about each request a specific client made:

```
struct request_rec {
    ...
    /** The connection to the client */
    conn_rec *connection;
    ...
    /** First line of request */
    char *the_request;
    ...
```

```
    char *useragent_ip;
};
```

With the help of certain known and fixed values also stored in the struct, our service can rapidly locate each stored client session in the raw memory with the help of appropriate regular expressions. It overwrites identified sensitive data with anonymous data which has exactly the same length as well as the same characteristics. For example, if an IP address in the *request_rec* needs to be anonymized, the service replaces the stored string with a new string having the same length, but also being a valid IP address. This is an important step since our architecture should not corrupt the cloned Apache2 process and its further execution. In particular, we replace information like the type of the performed request (e.g. GET or POST), the request string and the source IP address in every *request_rec*. Of course, the stored session of the attacker is not replaced with forged anonymous data. Moreover, depending on the setup, Apache2's *SSLConnRec* and *modssl_ctx_t* struct can be anonymized using the same strategy.

5.2.2 *Results of an Implementation*

We implemented the security service and the redaction procedure as a final step before the cloned process is restored by CRIU. In order to accelerate the proposed memory redaction procedure, we only analyze specific regions of Apache2's memory. By investigating Apache2's source code, we could narrow down the memory regions in which the target information is stored to the heap, the stack and anonymously mapped memory regions.

Extracting an attacker from the original Apache2 process into a cloned and isolated Apache2 process causes performance overhead. In our evaluation, we triggered the honeypatch with malicious requests and measured the overhead of individual requests. The proposed memory redaction strategy operates rapidly since it exploits the fact that Apache2 stores client session data in known and well-defined structs. Our evaluation showed that malicious HTTP requests cause a constant overhead of approximately 0.25 seconds in comparison to benign requests. However, this overhead only impacts detected attackers and does not influence other clients. The overhead is minimal and makes our proposed security service feasible for realistic environments, because the cloning, redaction and redirection procedures operate fast and this way they do not interrupt actions of an attacker nor they are recognizable. This section briefly showed how dynamically deployed honeypot environments can be realized on a more fine-grained and resource saving way with the help of software virtualization in the userspace of an operating system.

In this chapter, we proposed two different transparent security services which can be used to analyze ongoing attacks.

First, we proposed the extraction of a low-interactive honeypot VM from a customer VM (Section 5.1.2). We detect ongoing attacks in initial phases and rapidly extract a honeypot VM from a customer VM which is the target of the attack. The deployed new honeypot VM provides the same network applications in exactly the same up-to-date configuration. This way, the impact of the detected ongoing attack can be monitored and vulnerabilities which would have threaten the original VM can be revealed and analyzed while sensitive data remains secured. The architecture deceives and misleads an attacker but does not influence the normal work-flow of the original customer VMs. Based on a defined reporting format, cloud customers can learn from attacks and discover unknown vulnerabilities. We showed the design of the architecture, we presented details about the implementation and we evaluated the extraction and monitoring procedures. The evaluation showed that timely extraction of a new honeypot is feasible and realistic.

Second, we assumed an intruder who has already gained access to a customer VM. Once the intruder has been detected, we create a high-interactive honeypot VM from the original compromised VM and "extract" the intruder during run-time (Section 5.1.3). We implemented a VM live cloning process which removes sensitive data from the VM's volatile memory, inserts administrative data and blocks sensitive files on the VM's storage "on-the-fly". The cloned VM is finally transformed into a unique and custom-made high-interactive honeypot VM by deploying modified shell tolls in order to maintain the intruder's illusion to be still located in the original VM. During the extraction procedure, established connections of the attacker as well as of potential users logged-in in parallel are maintained. The evaluation showed that we can create a new high-interactive honeypot VM in approximately 6 seconds and periodically retrieve various information by transparently monitoring different points on the honeypot VM in steps of 1.6 seconds. The proposed service aggregates the collected information and generates a report for the original VM's owner in order to give a quick overview on the incident. It also analyzes the intruder's intentions. With our high-interactive honeypot service, new dimensions can be reached in terms of fooling attackers and researching their malicious actions and intentions.

In the last security service of this chapter, we briefly proposed a more light-weight approach in order to transparently analyze ongoing attacks in a cloned and isolated process environment (Section 5.2). We created separated virtualized Linux containers within a userspace and used them to deploy a replica of a process which we created and

restored during run-time. An attacker is detected with the help of honeypatches and redirected to the cloned process which has been redacted by removing sensitive data from its memory. The cloned process is monitored and the course of actions performed by the attacker is analyzed without interrupting the original process. We implemented the service for the Apache2 web server and an evaluation showed the feasibility of our approach since there is only a minimal overhead.

6 TRANSPARENT ATTACK MITIGATION

In the previous Chapters 4 and 5, we proposed transparent security services which detect and which analyze ongoing attacks in the IaaS cloud. In this chapter, we introduce two different transparent cloud security services which mitigate attacks. The first service dynamically isolates and securely stores sensitive data "on-the-fly" (Section 6.1) and the second service computes and implements custom-made security settings for the applications of cloud customers (Section 6.2). Both services operate in a pro-active fashion and use active VMI (Section 2.4) in order to manipulate the data of customer VMs during run-time.

6.1 DYNAMIC ISOLATION OF SENSITIVE DATA

As already mentioned in Section 1.2, flaws in software, for example in the memory management, are present in the cloud in the same manner as in traditional computer networks. Attackers can remotely exploit software vulnerabilities, for example in the network applications of customers, and this way circumvent authentication mechanisms in order to get access to the operating system and to spy on critical or sensitive data. Especially, stolen data like passwords or secret keys can be used in further attacks and cause damage on a large scale. For example, the famous Stuxnet [24] malware used digital certificates stolen from the systems of a hardware manufacturer in order to insert correctly signed driver modules into infected operating systems. Its successor, a malware called Flame [75], even used stolen digital certificates issued by Microsoft. Obviously, small sensitive data can be a worthwhile target for Cyber attacks.

In this security service, we focus on securing sensitive data located in the volatile memory of customer VMs. Our approach is based on a fact which greatly helps to develop a mechanism mitigating the probability of sensitive data being stolen: Sensitive data is in the majority of cases only required for small periods of time, while most of the time it is stored unused in the volatile memory of a running process, ready for access. A process uses the sensitive data once specific events are triggered which finally require the sensitive data (for example an authentication sub-procedure). Unfortunately, successfully executed attacks can bypass permission mechanisms of operating systems and simply retrieve the sensitive data from the process.

As a solution to this problem, we propose a transparent security service which securely stores known sensitive data in a co-resident

trusted VM (dom0) and only temporarily inserts the data into a customer VM to the exact location of use as long as it is required – and not longer. In particular, sensitive data is only inserted into the customer VM's volatile memory if certain previously defined integrity conditions could be verified. This way, the service greatly minimizes the probability of sensitive data being stolen by an attacker who successfully exploits vulnerabilities in order to bypass permission mechanisms. The proposed service benefits from the transparent perspective enabled by virtualization technologies (Section 2.3) and uses fine-grained VMI (Section 2.4). In comparison to the proposed security services of Chapter 4 and Chapter 5, VMI is this time not only used to transparently read certain information from the memory but also to write data to specific locations during run-time.

6.1.1 *Architecture of the Isolation Service*

The security service operates on the administrative VM (dom0) in a Xen hypervisor setup (Section 2.2). It dynamically isolates sensitive data of a customer's VM which co-residently runs on the same hardware component. To reach this goal, certain information need to be defined. The process name which uses the sensitive data (and runs on the customer's VM) and a characteristic fingerprint of the sensitive data to find its exact location need to be known. Furthermore, the service requires the definition of an event which triggers the temporary insertion of the removed sensitive data into the customer VM. For example, this event can simply be a new incoming TCP connection to a specific port on the operating system. Finally, a list of files and processes need to be defined. The integrity of these files and processes is verified by the service before insertion is allowed.

Basically, the proposed security service operates in seven sequential steps which are illustrated in Figure 33. The first step is an initial step and is executed only once.

In the *first* step, the service locates the allocated memory pages of the target process running in the customer VM. It searches for the exact location of sensitive data within these memory pages by generating a regular expression with the help of a characteristic unique fingerprint in the *second* step. Once the exact location of the sensitive data could be identified, the data is rapidly replaced by dummy data which has the same characteristics and exactly the same size. In this context, characteristics can be the format of the data or meta information of a file type. For example, if a key is isolated which includes certain specific meta information, the generated dummy key also needs to have valid meta information. The original sensitive data is securely stored in the administrative VM (dom0) which we assume as being trusted. Since the memory pages of the target process are continuously allocated and freed, the location of the dummy data

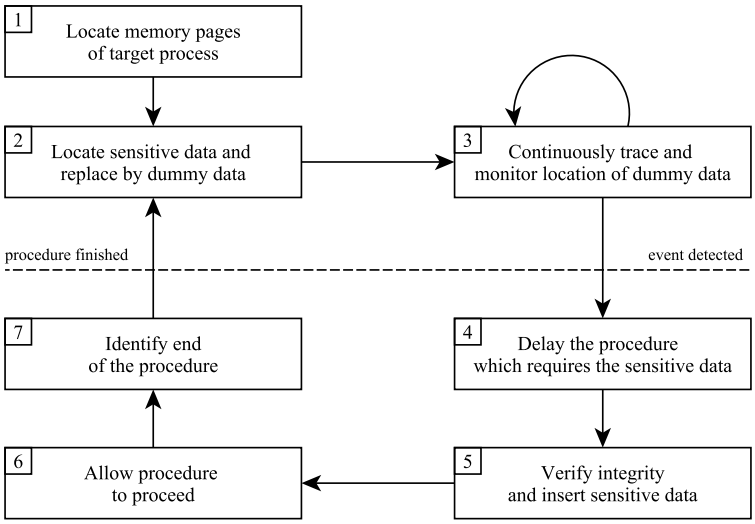

Figure 33: Steps of the security service which dynamically isolates sensitive data from a process running in a customer VM.

may change during the continuing execution of the process. Thus, in the *third* step, the proposed service periodically monitors the position of the inserted dummy data, again, with the help of the characteristic fingerprint. This way, the service continuously maintains the correct location of the dummy data in the volatile memory.

Once an event is triggered by a procedure of the process which immediately requires the original sensitive data, the service first delays the requesting procedure in step *four*. It verifies the integrity of the customer's VM before it temporarily replaces the dummy data by the original sensitive data in step *five*. In step *six*, the procedure of the process requiring the sensitive data is allowed to continue. Once the procedure is finished, the sensitive data is again replaced by new dummy data in step *seven* and the loop can start again waiting for the next requesting event.

In order to verify the customer's software platform before insertion is allowed, the service continuously retrieves certain information from four different points in the Xen hypervisor setup (Section 2.2). The service transparently retrieves information of currently running processes on the operating system using VMI (Section 2.4). It examines the VM's raw storage by extracting certain meta data of files using live storage forensic techniques (Section 2.5). Furthermore, it can retrieve information about the VM's network traffic from the virtualized network device as well as information about currently used memory and CPU performance from the hypervisor.

As already mentioned, verification of the operating system before the sensitive data is inserted is performed by verifying the integrity of stored files and running processes. These files need to be defined by the customer depending on it's operating system and installed applications. Integrity is measured by computing a hash over the data,

either over the data of a file located on the raw storage (Section 2.5) or over the data of the executable (ELF) code of a running process stored in the volatile memory (Section 4.1.1). Integrity is verified by comparing the hashes with white-listed hashes securely stored in the administrative VM (dom0). Once the integrity of the customer's VM could be verified, the service temporarily injects the original sensitive data into the location of use.

6.1.2 *Implementation and Evaluation*

We implemented a prototype of the security service for Linux. Its architecture is illustrated in Figure 34. The administrative VM (dom0) acts as the Trusted Computing Base (TCB) and uses a vTPM [10] coupled with a hardware TPM. It runs Linux and our security service which can transparently read from and write to the volatile memory pages of a co-resident customer VM on the same physical hardware. The customer VM runs processes requiring small sensitive data under certain circumstances (e.g. a secret key in an authentication procedure). The event triggering the insertion can be a new incoming remote connection to a specific port of the customer's operating system. In this context, an authentication procedure of a remote client can be easily delayed simply by delaying the network packets on the virtual network device.

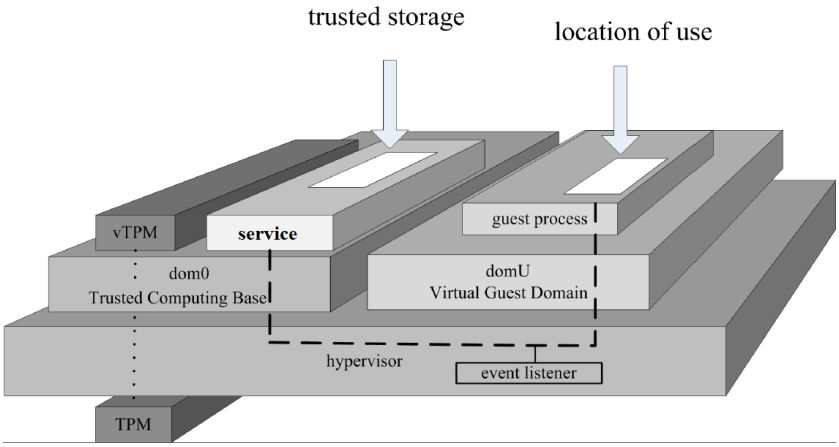

Figure 34: Architecture of the proposed security service which operates on a trusted VM (dom0) and dynamically isolates sensitive data of a process running on a co-resident customer VM.

We evaluated a scenario in which the proposed security service can be beneficially deployed. For this practical evaluation, we used an Intel(R) Core(TM) 2 Duo CPU with 3 Ghz and 4 GB of memory running the Xen hypervisor (Section 2.2). We deployed the service on the administrative VM running Ubuntu Linux 12.04.

The security service monitored a VM with 1024 MB memory running in parallel on the same physical hardware. The VM was vir-

tualized in HVM mode ("fully-virtualized") and run Ubuntu Linux 12.04. Additionally, the customer's VM run a SSH-server which allows secure remote access based on password authentication. In the evaluation, we treated the Linux password hashes (SHA512) as sensitive data which are continuously stored in the VM's volatile memory by the Linux Pluggable Authentication Module (PAM). The hashes are required for each Linux authentication procedure, but ending up in the wrong hands, the plain-text password could be ascertained with the help of a locally executed brute-force or dictionary attack in a high-performance hardware setup. Table 11 shows results of time measurements for the different procedures of our proposed service (100 runs).

procedure of the security service	**mean**	**std**
locate (original/dummy) password hash	3.10 s	0.22 s
replace (original/dummy) password hash	0.02 s	0.01 s
verify integrity of 32 files (locating and SHA256 hash)	1.68 s	0.23 s
verify integrity of 8 processes (locating ELF and SHA256 hash)	2.42 s	0.31 s

Table 11: Timings of the different procedures of the security service which dynamically isolates sensitive data.

Since locating the target password hashes and verifying the integrity of the operating system can run in parallel, an average SSH login procedure secured with our transparent security service took approximately 4.1 seconds while a standard SSH login procedure without our service took approximately 1.2 seconds in our setup. The proposed service causes an acceptable overhead, since it only delays the establishment of a new connection for approximately 3 seconds before the login credentials are requested from the new user. No further network packets of the same connection need to be delayed. By storing the hashes in a secured trusted environment and only inserting them once the integrity of the system is measured and only for a very short period of time for an authentication, the service greatly mitigates probability of the hashes being stolen. Most importantly, the proposed service operates generically which means every operating system and every application can be supported.

In the second transparent security service in the field of attack mitigation, we again use active VMI in order to modify the data of processes operating in customer VMs. We focus on the continuous improvement of security-related configuration settings of applications. Configuration settings which are not well-chosen, not correctly applied or too weak can lead to security breaches and loss of sensitive information. For example, allowing short passwords while not limiting the number of login attempts makes systems vulnerable to brute force or dictionary attacks. Often, customer-selected security settings are lax and do not fit well into the appropriate scenario. On the other hand, strong and static security settings can be perceived as being cumbersome since they can drastically decrease usability. In the worst case, they can even cause users to become annoyed and motivated to disable or circumvent the settings. For example recall the famous anecdote about people writing down complex passwords on a memo mounted on the computer display.

For this work, we analyzed configuration settings of 28 popular Linux networking applications. An average application has 43 different settings while 17 of these settings are related to security characteristics (40%). This means, if a VM of a customer runs three different networking applications, on average more than 50 security settings can be adapted to the setup and its specific properties.

In this context, we introduce the term "custom-made" settings. For example, the optimal limit for the maximal allowed length of requests to a Web server varies depending on the type of content a Web server provides and depending on the tasks it typically processes. Selecting the same limit for different Web servers of different customers will not be optimal even if these Web servers are exactly the same application – the properties of the target specific setup in which a Web server operates need to be included in the selection process. By analyzing characteristics of each Web server's setup, optimal settings can be derived. We call these setup-specific optimal settings custom-made settings.

Moreover, depending on how a setup is changing over time, if new software is installed or removed, if the number of users or the ways in which installed applications are used varies, custom-made security settings can change over time as well. Maintaining an overview in this medley of settings is a complex and time-consuming task in which failures can occur. For example, SSL/TLS misconfiguration attacks have been presented where incorrect settings result in potential Man-in-the-Middle attacks due to peer hosts not being verified[1]. Obviously, an automated approach for continuous hardening of applications seems to be a promising solution.

1 http://www.unrest.ca/peerjacking

In this section, we present a pro-active security service which autonomously and transparently improves the security settings of applications running in the VMs of customers. The service retrieves custom-made settings for each application from the corresponding setup by analyzing characteristics and properties, for example by investigating an application's network traffic or its log files. The service autonomously deploys these settings into the running application unbeknownst from the perspective of the VM and without the need to restart the application. The service can periodically adapt security settings and dynamically change them depending on occurred events or depending on changes in the characteristics of the customer's setup over time. Furthermore, the proposed security service can also change the settings promptly, depending on different tasks an application handles. This way, it can continuously provide optimized settings for an application in its unique scenario. We call this approach "hot-hardening". In particular, we define hot-hardening as the *continuous and transparent adaptation of security-related configuration settings of an application depending on properties and characteristics of its setup*. Hot-hardening can be provided as a cloud security service in an IaaS cloud computing scenario (Section 1.5).

6.2.1 *Custom-Made Settings – An Example*

Usually, an application comes with default settings stored in a configuration file which was created during installation. Suppose an application provides a login prompt on the network and only allows L_n failed login attempts until a connection is terminated and the source is banned for a fixed period of time. The default setting of L_n is 6 and the representation of L_n in the application's source code is an integer data type that is restricted to allow values in the range of 1 to 16. First, assume the application runs on server S1 which provides accounts for a large amount of users and which is not protected by a firewall which means everyone on the Internet can connect. Second, assume exactly the same application running on server S2 which provides accounts only for a few users who log-in rarely and S2 is protected by a firewall only allowing connections from within a sub network. Based on the properties of S1, a lower value for L_n than the default value can be recommended for S1, because the occurrence of brute force attacks against the login prompt will be higher.. Furthermore, the larger amount of user accounts on the system also increases the probability of some accounts being protected by weak passwords. On the other hand, the properties of the scenario in which server S2 operates result in less brute-force attacks because only connection from within the same network are allowed. Nevertheless, the failed login attempts can be high, because the users only log-in rarely which means, on average, they need more attempts because they may have problems to

remember their passwords. In this case, a higher value for L_n can be recommended that increases usability without decreasing security. As shown in this simple example, the same application can have different optimal settings depending on the setup in which it operates. In this section, we propose a pro-active security service which derives custom-made settings for applications running in a specific setup and being subject to unique surroundings and properties.

6.2.2 *Related Techniques: Hot-Patching*

Hot-patching is an existing technique for transparently updating a target application's binary code. Hot-patching was developed in order to minimize application and server down-times, as it applies patches to an application during its run-time. In particular, this technique is of interest when administrators want to deploy regularly released security patches. Huang et al. [36] developed an autonomous hot patching framework which can automatically identify the cause of a failure and patch the binary code of Web-based applications. However, hot-patching can also be risky since the system can run into an inconsistent state. To solve the consistency issues, Ramaswamy et al. [67] proposed a method for hot-patching ELF binaries which supports synchronized global data. In their work, they used Patch Objects which are patches encoded as ELF relocatable object files that can be automatically created and applied to a running process. Payer et al. [62] developed an update system that provides hot-patching by integrating dynamic patches with a sandbox based on dynamic binary translation. In a prototype implementation, they could patch 45 of 49 bugs on the Apache Web server at run-time. On the contrary, our new proposed technique called hot-hardening focuses on security settings of applications and not on executable code.

6.2.3 *Architecture of the Hot-Hardening Service*

In this section, we describe the different techniques used to transparently change settings of applications during run-time. We introduce application-dependent hot-hardening templates used by our service and the strategies for setting improvement.

6.2.3.1 *Hot-Hardening Procedure Overview*

The proposed security service runs in an isolated and separated VM equipped with our software. The service is privileged and can access the hypervisor layer as well as all the memory addresses on the physical hardware component. The service runs co-located with multiple customer VMs and can periodically hot-harden their appli-

cations with our proposed techniques and strategies. A simplified overview of a hot-hardening procedure is illustrated in Figure 35.

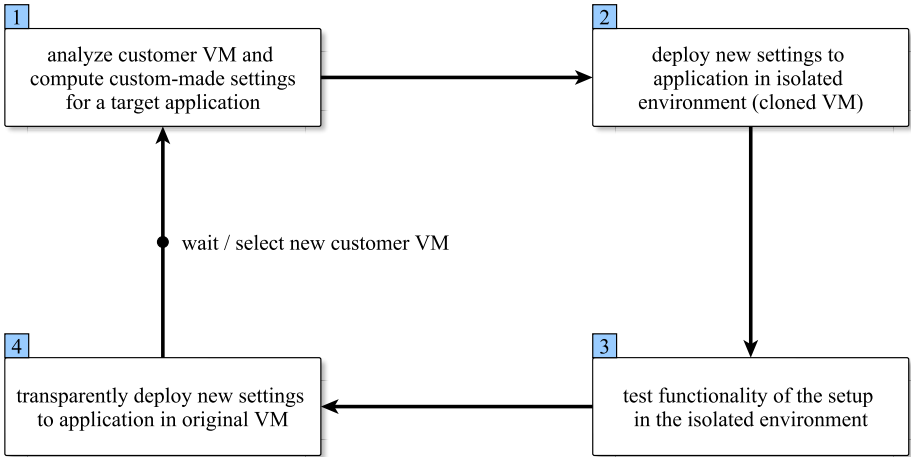

Figure 35: Simplified overview of the steps in a hot-hardening procedure performed by the proposed security service.

In the first step, the service discovers the operating system of the target VM and its running applications. For each operating system, there is a set of applications that the hot-hardening service can recognize and hot-harden. For supported applications, the service analyzes the properties and the status of the setup and computes custom-made settings for that application. In the second step, the service locates and deploys the new settings to a clone of the target VM in an isolated environment. This way, the settings can be tested and the execution of the original target customer VM is not disrupted. In the third step, the service verifies the correct execution of the application as it runs in the cloned VM; and finally, in step four, it transparently deploys the new settings to the original target VM. Because custom-made settings can change over time, the service repeats the hot-hardening procedure in certain intervals.

Since retrieving custom-made settings and testing the settings is a complex task requiring different strategies as well as semantic knowledge about the settings and their characteristics, we equip our service with application-dependent hot-hardening templates – one template for every supported application. The service can be equipped with multiple templates to support various applications. A template needs to be manually created and defines properties of an application and characteristics of its settings which are supported for hot-hardening. For each supported setting, the template defines two modules that are used by the service: A *get*-module used to retrieve and compute a custom-made setting from a setup and a *test*-module executing steps of a well-defined strategy in order to test a setting and its effects. The modules use setting-dependent information transparently retrieved from a target setup.

6.2.3.2 *Discovery and Hot-Hardening Templates*

In the first step, the service investigates which applications are running on a customer's VM by transparently examining the file system on the raw storage (Section 2.5) as well as open ports on the network. For discovery, the service also uses third party tools like the popular portscanner Nmap[2] which offers a database of fingerprints of more than 2000 well-known services. The service can also transparently locate the executable code (ELF) of running applications in the customer VM's volatile memory and compute hashes over this data for comparison with hashes of know applications in order to discover applications and their exact versions. If an application could be discovered for which the service maintains a hot-hardening template, a hot-hardening procedure is initated. Figure 36 shows an example template for a fictitious application called *mysrv*.

The template contains the name and the version of *mysrv* which is used to match the right template to a discovered application. The template provides the path to *mysrv's* configuration file located on the customer VM's raw storage. The service needs to read the configuration file in order to retrieve *mysrv's* currently deployed settings. Since not all settings of an application are related to security characteristics or suitable for a hot-hardening procedure, they need to be manually selected and defined in the template. However, some applications may not be suitable for hot-hardening because they do not have any settings or they only use settings in their start-up phase and do not store them or read them again during their operation. For the fictitious application *mysrv*, three different settings are supported for hot-hardening:

- Setting s01 can be set in a range of $0-100$ stepwise changed by 10. The given direction "low" defines that the security characteristics of *mysrv* increase the lower s01 is set. There is a module *get01.py* available that derives a custom-made setting for s01 from a target customer setup. Furthermore, there is a module *test01.py* available which can test the effect of s01 in a setup. In a real application, this setting could be for example a limit on the number of allowed requests.

- Setting s02 can only be set in a range of $0-2$ stepwise changed by 1. A lower value is beneficial for the security of *mysrv*. There is no module available to derive a custom-made setting for s02 which means the setting is independent from a setup. However its effect can be tested with the module *test02.py*. In a real application, this setting could be a string, for example defining the log-level of an application (high, medium, low). Usually, options which are defined as strings are stored as an integer value in the application's volatile memory.

2 http://nmap.org

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
</setting>
<setting name="s02">
        <range>0-2</range>
        <steps>1</steps>
        <direction>low</direction>
        <testModule>test02.py</testModule>
</setting>
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
</setting>
<instruction>s01|0|s03|*|s02</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>heap</location>
</template>
```

Figure 36: A Hot-hardening template of a fictitious example application which we called *mysrv*.

- Setting s03 has a possible range of $1024 - 4096$ while it can be stepwise changed by 128 and higher settings are beneficial for the security of *mysrv*. There is a module *get03.py* that retrieves the custom-made setting for s03 and a module *test03.py* that can test its effect. In a real application, this setting could be for example the length of a cryptographic key.

In particular, the proposed security service does not just select and set the best possible (highest or lowest) setting in a hot-hardening procedure. It uses the *get*-modules to find the best setting for a specific customer setup in the allowed range according to the template. Furthermore, the template defines an instruction how to generate a pattern which is used to locate the data structure that stores the currently deployed settings in *mysrv's* volatile memory. In this context, the data structure, the order of the stored settings and possible

content in-between them is important. In order to define an instruction for pattern generation, manual investigations of the application's source code is required. The template defines a memory address region in *mysrv's* memory in which the data structure of the settings is stored in order to accelerate the localization (in this case heap). Finally, it defines the different practical steps required in *mysrv's* hot-hardening procedure which are explained in detail in the following section.

6.2.3.3 *Transparent Setting Deployment*

Transparently changing settings on a running application is facilitated by the underlying virtualization technologies that allow the service to use different transparent techniques on a customer VM. First, the service uses active VMI to read and modify a running application's volatile memory (Section 2.4). Second, the service uses live storage forensics on the VM's raw storage and this way retrieves data of stored files. Settings are automatically replaced by the service unbeknownst from the perspective of the customer VM and without the need to interrupt an application's operation. In particular, the service deploys new settings by executing up to three different steps illustrated in Figure 37:

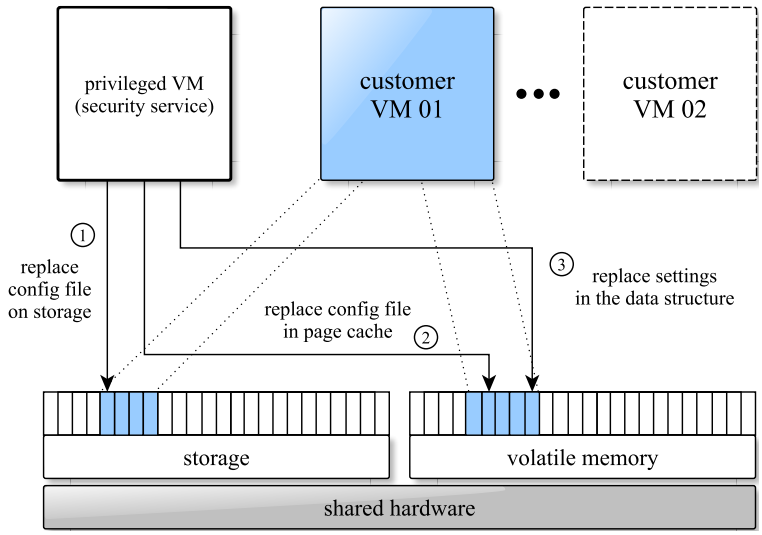

Figure 37: The hot-hardening security service uses three different practical steps to transparently insert new settings into a customer application during run-time.

1. The service locates the configuration file of the application by transparently examining the file system's meta data on the customer VM's raw storage. Once the corresponding sectors are located, the sectors are overwritten with new content having the same valid structure but containing the new settings. Replacing the configuration file on the storage in the first step pre-

vents automated recovery of old settings once the application is restarted or during run-time by reloading them from the file.

2. The service locates and replaces a cached copy of the application's configuration file in the operating system's page cache with the new content. The page cache helps applications to retrieve stored data faster by keeping recently requested data in the volatile memory. Replacing potentially cached content of the configuration file prevents automated recovery of old settings.

3. The service locates and replaces the currently used configuration settings in the application's volatile memory. To reach this goal, data structure alignment and endianness of the settings need to be considered. There are a lot of ways in which an application can process a configuration file and store its settings in memory. This depends on the programmed data structure and on the used data types. Often, settings are summarized and stored in a struct data type, in some cases settings are stored independently. However, there are cases in which settings are not stored in memory and read from the file each time they are required. In order to locate the settings in memory, the service uses the *instruction* of the template to generate a pattern and searches in the memory region defined in the template.

In same cases, depending on the target application, only the steps 1 and 2 are required, because the application does not store settings in volatile memory. Sometimes only step 1 is required, because there is no page cache as well as there are no settings in the application's volatile memory. Sometimes only step 3 is required, because there is no configuration file but there are settings in the application's memory fetched from other sources like for example system variables or command line arguments. The template defines which steps are required for a specific application's hot-hardening procedure.

6.2.3.4 *Isolated Setting Testing*

Changing settings with the help of VMI during an application's runtime can lead to interruptions in the application's operation, unexpected results and to misleading entries in log files. In order to avoid these problems on a customer VM, the service first clones the VM and deploys and tests the new settings on the clone. Once better working settings could be found, the service finally deploys them on the original customer VM. As introduced in Section 2.3, VM live cloning [81] is a modified live migration [18] process which creates a synchronous replica of a target VM during its run-time. The procedure is illustrated in Figure 38. A virtual snapshot of the virtual storage of the customer VM is created and the volatile memory is copied in multiple subsequent iterations.

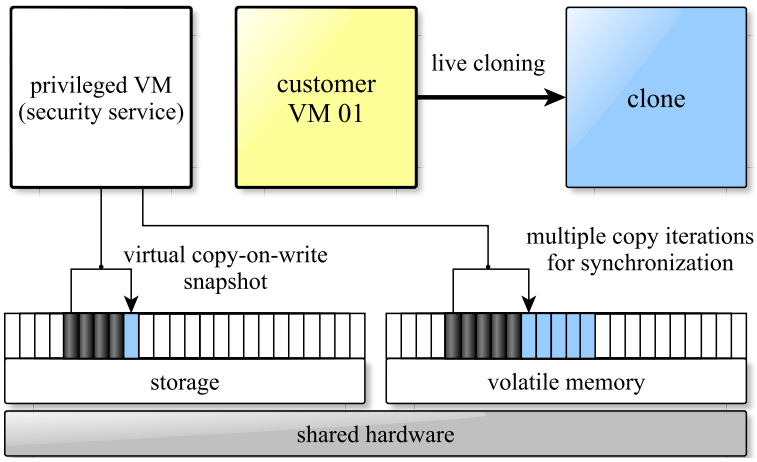

Figure 38: The live cloning process triggered by the proposed hot-hardening security service in order to test new settings on a clone.

If the service changes multiple different settings or settings of different applications on the same customer VM, dependencies of settings can play a role. For example, application *A* can operate together with application *B* and changing a setting of application *A* can increase their mutual workload which can overload application *B*.

In order to test new settings and to detect potential unknown dependencies or any other problems, the service evaluates the effect of each new setting by executing a specific setting dependent *test*-module. Each *test*-module which tests the characteristics of a setting i returns a numerical result R_i that can deviate from a known theoretically expected result R_i^e depending on the currently deployed setting. If multiple settings are changed, the service first deploys the best settings as defined in the template and tests the effects on the setup. If the results of the test of a new setting deviates from its expected effect ($R_i^e - R_i > 0$), the service stepwise reduces the setting according to the defined steps and direction in the application's hot-hardening template. In multiple rounds, the service evaluates the tests of all changed settings and tries to find the best group of settings. This way, the service can reveal if changing a setting influences the effect of another setting or if a new setting can not be deployed. The testing strategy is illustrated in Figure 39. In the *first* step, the service changes a single setting and tests the effect of this setting in a *second* step, but tests also the effects of all other settings which have not been changed in a *third* step. It may happen that an application terminates during the hot-hardening procedure. In this case, the service creates a new clone. Once the best working group of settings could be found, the service terminates the clone and finally deploys the new and tested settings on the original customer VM.

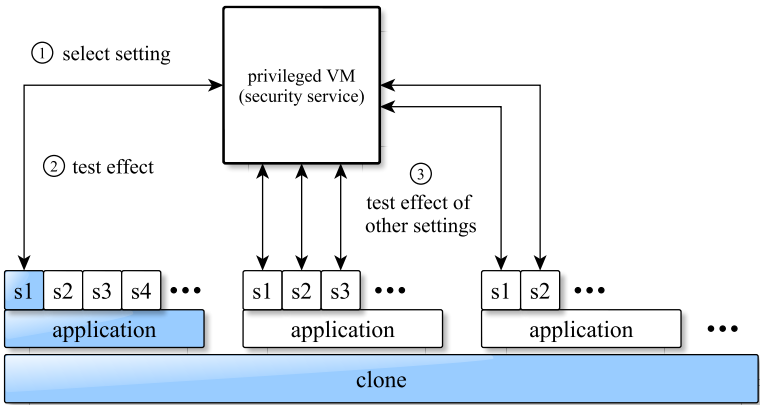

Figure 39: The service stepwise revokes the settings by replacing them with reduced settings once deviations in the effects could be detected.

6.2.4 *Dynamic Child Adaptation (DCA)*

Up to this stage, our proposed pro-active security service transparently changes settings, tests the new settings in an isolated environment (cloned VM) and deploys them on a customer VM. In order to define the semantics of settings and to allow the service to work autonomously, we introduced application-depending templates. Furthermore, the proposed security service uses setting-depending modules to retrieve a custom-made setting by analyzing a target customer setup and to test the effects of a new setting on a setup. As previously shown in Figure 35, the proposed service can periodically repeat a hot-hardening procedure on the same customer VM and this way continuously adapt settings which may become suboptimal over time since the setup or its scenario changed.

However, sometimes it is beneficial to allow the service to dynamically change settings depending on events. In this context, an event can be a new connection or a new task the application handles. Often, an application creates temporary child processes to handle different tasks separately, for example it creates a new child process for each new network connection. Sometimes, an application also assigns a data structure of settings to the child processes or it creates a new child process with its operations depending on currently deployed settings of the main process. This allows our proposed service to modify the settings of new child processes separately. The service can deploy different settings for each new child process depending on certain events and on properties.

We call this fine-grained and dynamic hot-hardening procedure Dynamic Child Adaptation (DCA). In particular, a property of a child process, for example of the connection it handles, can lead to an adaptation of the child's security related settings. If an application has been hot-hardened once, the service can continuously monitor the customer VM and adapt certain settings of the application's child

processes – if supported. As an illustrative example, we extend the hot-hardening template of the fictitious application *mysrv*. We add information for which settings DCA is supported and which events or properties lead to which adaptations (Figure 40).

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>127.0.0.1</src></event>
                <set>100</set>
        </dca>
</setting>
...
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1024</set>
        </dca>
</setting>
...
```

Figure 40: Extensions to *mysrv's* hot-hardening template in order to support Dynamic Child Adaptation (DCA) for s01 and s03.

The new template allows our service to use DCA for the two settings s01 and s03. Once a child process handles a connection from the own system, setting s01 is set to 100. Once a child process handles a connection from the same sub net, setting s03 is set to 1024. This way, security settings can be relaxed under certain circumstances in order to increase usability or security characteristics can be dynamically amplified. Furthermore, if DCA is used, the settings are directly inserted into the customer VM without testing them on a clone first. However, DCA is only supported after an application has already been hot-hardened, which means changing settings was already tested.

A DCA procedure which adapts child processes is illustrated in Figure 41. In this case, a new incoming connection is detected and analyzed by the service before it reaches the customer VM. The service retrieves the source IP address and several other information from the new connection. This can be performed since communication is tunneled through the virtualized network device of the hypervisor (Section 2.2). If the properties of a new connection match the event definition in the new extended template, the connection is delayed until the settings of the application are adapted. After adaptation, the connection is allowed to proceed and the application now creates a specially adapted child process. Once the child was created, the service immediately revokes the settings in the application back to the custom-made settings which are effective for other clients. In this context, fine-grained connection management is required.

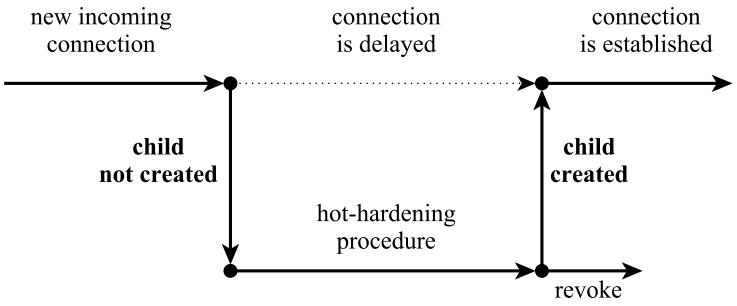

Figure 41: Subsequent steps in an example Dynamic Child Adaptation (DCA) hot-hardening procedure.

6.2.5 *Evaluation of the Hot-Hardening Service*

In an evaluation, we investigate hot-hardening procedures applied to popular applications and analyze the results. We use a Xen hypervisor setup (Section 2.2) with a Quad CPU (2.67 GHz), 8 GB memory and a customer VM with 4 GB memory. Both run Ubuntu Linux 12.04 in 64-Bit.

6.2.5.1 *Example: Apache2*

First, we investigate a hot-hardening procedure for the Apache2 Web server 2.2. Our Apache2 Web server is a production system hosting our Web page and giving access to a regularly used SVN repository. Without considering any Apache2 extensions, we identified 7 settings of the application which are related to security and which are suitable for hot-hardening. Apache2's configuration file is located in */etc/apache2/apache2.conf*. The settings are stored in Apache2's volatile memory as a *struct* data type called *server_rec* defined in the source code in *include/httpd.h*:

```
struct server_rec {
    ...
    /** Timeout, before we give up */
    apr_interval_time_t timeout;
    /** Interval we will wait for another request */
    apr_interval_time_t keep_alive_timeout;
    /** Maximum requests per connection */
    int keep_alive_max;
    ...
};
```

By default, Apache2 handles new connections in child processes which introduces the opportunity to apply DCA.

APACHE2'S HOT-HARDENING TEMPLATE: A fragment of the hot-hardening template for Apache2 can be seen in Figure 42. Seven different settings are supported. Their effects mitigate Brute-Force attacks, prevent malicious tools to scan for vulnerabilities and misconfiguration and avoid Denial-of-Service (DoS) attacks as well as the successful execution of exploits (for example based on large or unusual input). The supported settings are briefly explained in the following:

- s01 defines the log level ($0-7$) and is originally a string (emerg, alert, crit, ..., info, debug). The template limits the setting to seven levels $0-6$ which can be changed stepwise by 1 while 6 logs the most (info). Logging more details increases Apache2's security characteristics.
- s02 defines the time Apache2 waits to receive a request from a new connection before terminating it. The template defines a range of $10-300$ seconds for s02 which can be stepwise changed by 10. A low setting increases the security characteristics of Apache2.
- s03 defines the maximum number of requests allowed during a persistent connection before Apache2 terminates it. The template defines a range of $10-100$ for s03 that can be stepwise changed by 2, the lower it is set the better the security characteristics are.
- s04 defines the time for which a connection is maintained for a next request. The template defines a range of $2-30$ seconds, a lower value is beneficial for security.
- s05 defines a limit for the size of HTTP request lines in bytes. The template defines a range of $1024-8192$ while low is assumed as being better.

- s06 limits the size of request header fields in bytes and ranges between $1024 - 8192$.
- s07 limits the number of header fields in requests. It can be set in a range of $10 - 100$.

```
<template>
...
<setting name="s01">
        <range>0-6</range>
        <steps>1</steps>
        <direction>high</direction>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1</set>
        </dca>
</setting>
...
<setting name="s06">
        <range>1024-8192</range>
        <steps>512</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <testModule>test06.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>8192</set>
        </dca>
</setting>
...
<instruction>s01|*|s02|0|s04|0|s03|.|s05|0|s06|0|s07</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>anonymous</location>
</template>
```

Figure 42: The hot-hardening template of Apache2.

For s01 no *get*-module is available since this setting is assumed as being independent from a setup. For the settings s02 to s07, *get*-modules are available which can retrieve a custom-made setting from a setup. The effect of each setting can be tested with a *test*-module. The template also defines an instruction how to generate a pattern in order to locate the currently deployed settings (derived from the configuration file) in Apache2's volatile memory. In this context, the order of the stored settings and the content in-between them (for example other settings) is important. This information was previously derived from the source code. The template defines which practical

hot-hardening steps are required (Section 6.2.3.3) and the memory region in which the data structure of the settings is stored. In this case, it is stored in anonymous mappings created by mmap() with the *MAP_ANONYMOUS* flag. The template allows Dynamic Child Adaptation (DCA) for the settings s01 and s06. Setting s01 is dynamically set to 1 which means less events need to be logged if a client connects from our sub network. Setting s06 is set to the maximum of 8192 for clients connecting from our private sub network which allows them to perform longer requests with more header information.

APACHE2'S HOT-HARDENING MODULES: The template specifies six *get*-modules to retrieve custom-made settings for s2, ..., s7 from our setup. The computed settings depend on various properties, for example the workload of our system, other installed applications or the kind of data our Web server provides. Our Apache2 server hosts a Web page and a repository (SVN) that can be remotely used by employees who mostly connect from within our private network. This is a special property of our setup which actually leads to a frequent occurrence of multiple GET or PUT requests in a row caused by users rapidly synchronizing SVN projects consisting of multiple files. The *get*-modules compute an average value of events occurring on the setup which are affected by a setting, they analyze the distribution of events in log files or they investigate network traffic of the application. The *get*-modules select the best possible value as custom-made setting according to the definitions of allowed steps and direction in the template.

- get02.py analyzes network traffic and the timings of new connections and their first request. In our setup, the mean is 2.89 ± 4.01 seconds which leads to a custom-made setting of 10 seconds (default is 300).
- get03.py analyzes network traffic and investigates the average number of requests performed per connection (Figure 43). In our setup, there is a second peak in the distribution caused by users synchronizing multiple files with our SVN repository. This is a special property of our setup and leads to a custom-made setting for the limit of requests per connection of 42 (default is 100).
- get04.py analyzes timings between multiple request on a persistent connection (Figure 44). It calculates a custom-made setting of 18 seconds for our setup (default is 30).
- get05.py analyzes the length of requests retrieved from the access log file (Figure 45). It calculates a custom-made setting of 1536 bytes for our setup (default is 8190).

- get06.py analyzes the length of the headers in requests from the Apache2 log file. It calculates a custom-made setting of 1024 bytes for our setup (default is 8190).
- get07.py analyzes the numbers of fields used in headers in requests. It calculates a custom-made setting of 30 (default is 100).

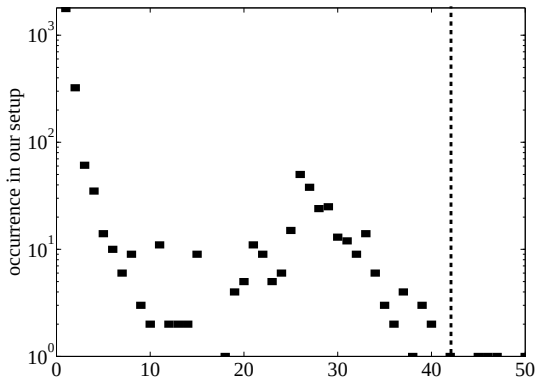

Figure 43: Distributions of requests per connection extracted from our setup by the *get*-modules which use this data to derive a custom-made setting for s03 (marked as dotted line).

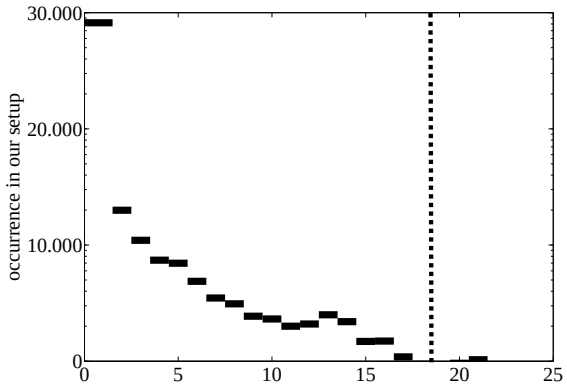

Figure 44: Distributions of time between requests [s] extracted from our setup by the *get*-modules which use this data to derive a custom-made setting for s04 (marked as dotted line).

The *get*-modules consider the special properties of our setup and derive custom-made settings for Apache2 which increase security while only minimally affecting our work-flows. The *test*-modules actively perform tests on the setup. Each *test*-module analyzes the effect of a currently deployed setting. For example, they establish new connections to Apache2 to verify expected timeouts, limits or other characteristics caused by the new settings. In this context, the *test*-module

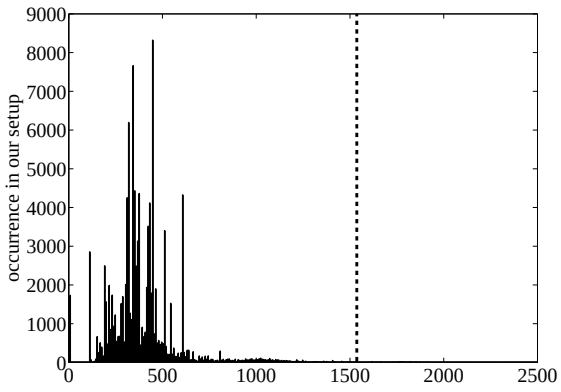

Figure 45: Distributions of length of requests [bytes] extracted from our setup by the *get*-modules which use this data to derive a custom-made setting for s05 (marked as dotted line).

for s01 inspects the information that is logged while it actively performs requests, other *test*-modules analyze the timeouts for s02 and s04 while maintaining a connection and the module for s03 tests how many requests are possible on a persistent connection. The modules which test s05, s06 and s07 perform different kinds of requests (length and header fields) to evaluate the corresponding new settings.

APACHE2'S HOT-HARDENING PROCEDURE: In this section, we investigate the hot-hardening procedure in detail after the custom-made settings have been computed. First, the service clones the customer VM before applying and testing any new settings. In our setup, live cloning (Section 2.3) of a running VM with 4 GB of memory into a synchronous replica can be performed in 17 ± 2 seconds on average. The timing strongly depends on the current workload of a VM. In order to reduce the time required for cloning, prepared clones can be maintained or clones can be created using copy-on-write (CoW) methods for both volatile memory and storage which enables the creation of a clone in the range of milliseconds [81].

In the first step, the service identifies the currently deployed settings by locating and reading the corresponding sectors of Apache2's configuration file on the raw storage. In general, *struct* members are stored in the order they are declared (C99 standard) and if necessary, padding is added before each *struct* member to ensure correct alignment. Setting s02 and s04 are stored as long data types representing microseconds. The other settings are stored as integer data types. Furthermore, the order of the byte representation of s02, s04, s05 and s06 has to be adapted according to Little-Endianness. With the help of the known current settings and the instruction defined in the hot-hardening template, the service generates a pattern to locate the deployed settings in Apache2's volatile memory. Table 12 shows the con-

version procedure of characteristic values. The generated characteristic pattern in hexadecimal representation is 04(.)*?00a3e111(0)*?80c3c901(0)*?64(0)*?fe1f(0)*?fe1f(0)*?64. It allows padding with zeros between successive settings and arbitrary content between settings which have other settings in-between.

Allowed time between a new connection and the first request of this connection	300 s = $300 \cdot 10^6$ µs = 11e1a300
Allowed time for maintaining a connection between two subsequent requests	30 s = $30 \cdot 10^6$ µs = 1c9c380
Allowed length of the HTTP request string in bytes	8190 bytes = 1ffe
Allowed number of requests per established connection	100 = 64

Table 12: Conversion and computation of settings into hexadecimal representation used to locate the corresponding data structure of the Apache2 settings in the volatile memory.

Retrieving custom-made settings from our setup using the seven different *get*-modules takes 121 ± 4 seconds based on an anonymized network capture of 62 MB and a log file of 24 MB. Finding the corresponding sectors of the configuration file on the storage and replacing the content with new content takes only 0.3 ± 0.1 seconds. Locating and replacing the content of the cached configuration file in the Linux page cache takes 5.2 ± 0.9 seconds. Generating the pattern according to the instructions of the template, locating the data structure of the settings in Apache2's volatile memory and replacing the currently deployed settings with new settings takes 2.4 ± 0.1 seconds. Once new settings are deployed, the service executes the setting dependent tests and analyzed the results in order to reveal unexpected deviations. Evaluating the seven tests takes 46 ± 8 seconds. Finally, a full hot-hardening procedure for Apache2 takes around 3 minutes in our setup without any manual interaction. This is a good result considering that the proposed service performs deep analysis on network traffic and log files and that the it also needs to wait during the execution of the *test*-modules in order to verify timeouts.

For DCA, no configuration files need to be replaced. The settings s01 and s03 are dynamically hot-hardened. Analyzing a new connection and transparently matching the source IP address to a new child on the VM via VMI takes 1.4 ± 0.2 seconds. Hot-hardening of a child once the triggering properties of a new client could be detected takes only 2.1 ± 0.1 seconds. Accordingly, a new connection which is the target for DCA needs to be delayed for approximately 3.5 seconds until the corresponding child is hot-hardened and operates under specific settings.

HOT-HARDENING APACHE2 WITH PHP5: Apache2 usually runs in a combination with several extensions. In this section, we introduce a template for PHP5 hot-hardening and let the service hot-harden Apache2 and PHP5 on the same setup. Interesting security-related settings of PHP5 are stored in a *struct* data type called *php_core_globals* defined in the source code in *php_globals.h*:

```
struct _php_core_globals {
    ...
    long memory_limit;
    long max_input_time;
    ...
};
```

The template for PHP5 provides five different settings which are related to security and supported by our security service:

- s01 defines the maximum amount of memory (MB) a PHP script can consume (default 128 MB).
- s02 defines the maximum time a PHP script can spend parsing a request (default 60 seconds).
- s03 defines the maximum size (MB) for uploads (default 2 MB).
- s04 defines the maximum level of nested requests, for example of an array[][][] (default 64).
- s05 defines the maximum number of GET and POST input variables (default 1000).

The modules are briefly described in the following: The *get*-module of s02 analyzes the timings between requests for PHP scripts and the resulting replies sent from the server. The *get*-module of s03 analyzes the Apache2 log for uploads and their sizes. The *get*-modules of s04 and s05 analyze the PHP code found in Apache2's public directory (default */var/www/*) for definitions of nested variables and their depth and the amount of variables accepted by scripts.

The hot-hardening service retrieves the currently deployed settings from the configuration file */etc/php5/apache2/php.ini* and generates a pattern according to the instruction given in the PHP5 template while taking into account data alignment and Little-Endianness. The settings are stored in the anonymously mapped memory region of the library libphp5.so used by Apache2. The resulting pattern is 0000008(0)?3C(.)*?000020(.)*?40(.)*?e803.

The service uses the *test*-modules to evaluate the new settings, in this case also by creating and inserting PHP code on the clone's storage. To reach this goal, the service creates specific PHP files in the

public directory of Apache2 and triggers the execution on the network in order to evaluate the PHP characteristics. The test01.py creates and evaluates a script that allocates up to 128 MB of memory when requested, test02.py creates a script that delays its execution to up to 60 seconds, test03.py creates a script that allows uploading data of arbitrary size, test04.py creates a script that allocates a nested array of an arbitrary size and test05.py creates a script that accepts a defined amount of input variables via a GET request. These tests can test the characteristics of the PHP5. Execution of the modules could be performed in 104 ± 3 seconds and insertion of new custom-made settings to PHP5 running in the clone while also replacing the stored configuration file and the file in the page cache could be performed in 8 ± 1 seconds. Custom-made settings for PHP5 retrieved from our setup can be seen in Table 13.

PHP5 setting	**default setting**	**new setting**
s_1	128	16
s_2	60	6
s_3	2048	1536
s_4	64	4
s_5	1000	25

Table 13: Previous settings and new custom-made settings used in PHP5's hot-hardening procedure.

In the first step, Apache2 is hot-hardened and the setting dependent tests are evaluated for both Apache2 and PHP5 according to the proposed testing strategy. In the second step, the service hot-hardens PHP5 and again evaluates the tests for both applications. Since the service evaluates two templates, dependencies can come into play. For example, during Apache2's hot-hardening, the evaluation of PHP5 test03.py detected an unusual deviation between the expected characteristics and the actually measured characteristics on the setup. PHP5 test03.py evaluates the maximum allowed size for uploads according to the PHP5 setting s03 which defines 2 MB in our setup before hot-hardening. After the service changed Apache2 setting s02 which defines the timeout for requests to the new custom-made setting of 10 seconds, the PHP5 test03.py could not upload a file of size 2 MB since the new timeout was too low for the upload. Following the proposed strategy, the service automatically revoked Apache2 setting s02 and deployed the next possible step defined in Apache2's template, which is 20 seconds. However, identifying a dependency between Apache2 setting s02 and PHP5 setting s03 caused two additional testing rounds which led to a hot-hardening time for Apache2 of 8 minutes instead of 3 minutes. Dependencies like this can occur when applications share resources or somehow work together like in this scenario. However, with the help of the strategy

and well-defined templates, dependencies can be revealed and settings can be successfully adapted. Finally, hot-hardening of Apache2 and PHP5 took approximately 11 minutes in our setup.

6.2.5.2 *Example: OpenSSH2*

Finally, we investigate hot-hardening of OpenSSH2 in detail. Our OpenSSH2 server is a production system having several users who log in regularly, also using the SCP and sFTP protocol.

OPENSSH2'S HOT-HARDENING TEMPLATE: The configuration settings of OpenSSH2 are stored in a *struct* data type called *ServerOptions* defined in *servconf.h*. The template for OpenSSH2 can be partially seen in Figure 46 and supports 8 different security settings.

```
<template>
...
<setting name="s04">
        <range>5-120</range>
        <steps>5</steps>
        <direction>low</direction>
        <getModule>get04.py</getModule>
        <testModule>test04.py</testModule>
</setting>
<setting name="s05">
        <range>1-6</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get05.py</getModule>
        <testModule>test05.py</testModule>
</setting>
<setting name="s06">
        <range>0-1</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <dca>
                <event><src>192.168.1.7</src></event>
                <set>1</set>
        </dca>
</setting>
...
</template>
```

Figure 46: The hot-hardening template for OpenSSH2.

The supported settings are briefly explained in the following:

- s01: Size of the key (protocol 1, default 768 bits).
- s02: Time after which the key is regenerated (protocol 1) in seconds (default 3600 seconds).
- s03: Different ciphers which are allowed (protocol 2) (e.g. aes128-cbc, blowfish-cbc, ...). The template defines three groups classified by strength.
- s04: The server disconnects after this defined time if the client has not logged in (default 120 seconds).
- s05: Maximum number of authentication attempts allowed before the connection is terminated (default 6).
- s06: Specifies if host-based authentication (together with public key) is allowed (default *no*).
- s07: Specifies if forwarding of X11 is allowed (default *no*).
- s08: Specifies if remote root login is allowed (default *no*).

The template defines the location of the configuration file on the customer VM's raw storage and an instruction and region used to locat settings in OpenSSH2's volatile memory. The template defines DCA for the setting s06, s07 and s08. Host-based authentication is only enabled for a back-up server (192.168.1.7) that daily and automatically transfers data. Accordingly, no other client is allowed to use host-based authentication. Forwarding of X11 is allowed only for clients connecting from our private network and remote root login is only allowed for one host used by the administrator. The OpenSSH2 server is dynamically hot-hardened which allows different settings under different circumstances which normally could only be statically deployed for all clients.

OPENSSH2'S HOT-HARDENING MODULES: The *get*-modules and their results in our specific setup are briefly explain in the following:

- get01.py investigates if protocol 1 is supported and which key size is used by default. It the suggests to use highest setting (2048).
- get02.py and get03.py investigate the status of the setup and propose 300 for s02 and 3 for s03, which means only the strongest cipher algorithms are supported.
- get04.py investigates a network capture and the authentication log file and analyzes the timings between the establishment of a new connection and a successful login. In our setup 7.31 ± 2.76 which leads to a suggested custom-made setting of 10.

- get05.py analyzes the average number of login attempts remote users need to successfully log into the system. These are 1.11 ± 0.39 in our setup which leads to a custom-made setting of 2.

The *get*-modules get06.py, get07.py and get08.py investigate the status and suggest the best setting if feasible. Table 14 shows the computed custom-made settings for our specific setup.

OpenSSH2 setting	default setting	new setting
s_1	768	2048
s_2	2600	300
s_3	12	4
s_4	120	10
s_5	6	2
s_6	1	0 (1 in DCA)
s_7	0	0 (1 in DCA)
s_8	0	0 (1 in DCA)

Table 14: Previous default settings and new settings used in OpenSSH2's hot-hardening procedure.

OPENSSH2'S HOT-HARDENING PROCEDURE: In our test-runs, the service clones the VM in 16 ± 4 seconds and uses the *get*-modules to retrieve custom-made settings in 47 ± 5 seconds Finding the sectors of the OpenSSH2 configuration file and overwriting the data on the raw storage takes 0.2 ± 0.3 seconds. Locating and replacing the file's cached content in the Linux page cache takes 5.4 ± 0.8 seconds. Generating the pattern according to the given instructions, locating the currently deployed settings in OpenSSH2's volatile memory and replacing these settings with the new custom-made settings takes 1.9 ± 0.2 seconds. The testing phase takes only 31.2 ± 2.2 seconds. Finally, the hot-hardening procedure of OpenSSH2 could be automatically performed by our proposed hot-hardening service in approximately 1 minute and 42 seconds.

After the hot-hardening procedure is finished, DCA is enabled, which means the service continuously monitors the customer VM and dynamically adapts the settings s06, s07 and s08 for new child processes depending on the IP address of a new client. DCA for OpenSSH2 can be performed by the service in 2.1 ± 0.2 seconds which means a new TCP connection establishment to the SSH port needs only to be delayed for that period of time until it can proceed. Other connections for which DCA is not enabled are only delayed in the range of milliseconds. In our scenario, DCA enables specific settings for our backup server which now can log-in via host-based authentication while this option is disabled for all other clients. In the same manner, X11 forwarding and remote root login is only allowed for

specific clients while maintaining stronger security settings for the others. DCA allows the definition of more fine-grained settings and can greatly increase security as well as usability.

6.2.6 *Limitations of the Proposed Strategy*

As already mentioned, the proposed security service can not hot-harden every application. Some applications do not store configuration settings in memory or they do not provide any configuration settings. Only applications which store settings in memory and which also regularly and periodically read and use these settings during their operation are suitable. Some applications only adapt their operation during the start and do not read the settings again during their operation. Additionally, not all security settings of an application are suitable for hot-hardening – they need to be well-chosen and defined in the proposed template. Especially, DCA is only applicable if the application assigns the settings to the child processes or at least adapts the operation of new child processes according to the settings. Furthermore, there is no guarantee that unknown dependencies of different applications can be detected in every scenario. The templates need to be well-defined and the security service should only hot-harden settings which effects can be reliably estimated.

6.3 SUMMARY

In this chapter, we first presented a transparent isolation security service which secures sensitive data of customer VMs by dynamically relocating the data to a trusted environment (Section 6.1). Right before the sensitive data is required, for example by an authentication procedure of a process, the service temporarily inserts the sensitive data into the exact location of use. The proposed service only inserts the data if the integrity of the customer's operating system could be verified. Vice versa, once the sensitive data is not longer required, the service immediately isolates the data again and replaces it with dummy data. Thus, the sensitive data is only temporarily inserted into the customer VM's memory for a very short period of time and only as long as it is required. This way, our proposed security service minimizes the probability of sensitive data being stolen with the help of attacks which bypass permission mechanisms. An evaluation showed that our strategy is feasible and can be beneficially deployed since it operates generically and causes only a small overhead.

Second, we proposed a pro-active security service which actively optimizes security settings of applications of customers (Section 6.2). Applying optimized security settings for different applications is a difficult and time-consuming task. We presented an autonomous security service that computes custom-made settings for a specific setup

and periodically improves security settings of different applications running on customer VMs. We called this strategy hot-hardening. The service uses transparent techniques in order to modify configuration files and data structures that store settings in an application's volatile memory. The operation of an application is not disturbed as well as no customer interaction is required. The service supports applications for which a template is defined and for which setting-dependent modules are available. Modifying the source code of an application is not required. Since optimal settings can change over time our service can periodically analyze the characteristics of a setup and continuously adapt the security settings. In order to support different setting for different tasks, we introduced Dynamic Child Adaptation (DCA) which enables the service to rapidly deploy specific settings for new child processes depending on properties of the task they handle. In an evaluation, we showed that the service can hot-harden three different popular applications (Apache2, PHP5 and OpenSSH2) in feasible time and can automatically improve their security characteristics.

7 CONCLUSION

In this work, we presented different new security architectures for an IaaS cloud computing scenario (Section 1.4). The proposed security architectures can be provided as flexible and scalable services to cloud customers according to the Security-as-a-Service (SECaaS) business model (Section 1.5). We used techniques like fine-grained VM introspection (VMI) (Section 2.4), VM live storage forensics ((Section 2.5)) and VM live cloning (Section 2.3) to design and implement our security services in a Xen hypervisor setup (Section 2.2). All the proposed security services operate transparently and run in a tamper-resistant environment isolated from the systems of the cloud customers (Section 2.1). The proposed security services use novel and unique strategies to analyze or modify data transparently retrieved from the hypervisor perspective during run-time.

First, we proposed three different security services in the field of transparent attack detection (Chapter 4). The first service detects unknown spreading malicious software by revealing the same propagating effect on an increasing number of different customer systems within the IaaS cloud network (Section 4.1). The second service detects phishing attacks aiming at customers and executed with the help of hitherto unknown spoofed web pages. It generates a color-based fingerprint from information directly retrieved from the raw volatile memory that is currently used by a browser application during its operation. Fingerprints are compared and spoofed web pages are revealed with the help of human perceptual similarity analysis (Section 4.2). The third security service detects Man-in-the-Browser (MitB) attacks by comparing the data of web pages passing the virtualized network device with the data of the web pages processed by a browser application in the volatile memory of a VM (Section 4.3).

Furthermore, we proposed two different security services in the field of transparent attack analysis (Chapter 5). The services precisely analyze ongoing attacks as well as they deceive and misdirect attackers with the help of honeypot-like environments. The first security service dynamically extracts low-interactive or high-interactive honeypots from customer systems which has the advantage that they run the same software in exactly the same configuration – but without sensitive data. Attackers can dynamically be redirected or even logged-in intruders can be extracted "on-the-fly" into isolated systems used for analysis (Section 5.1). The second security service shows how an attacker can be trapped on the process level by dynamically cloning a process which is under attack instead of cloning a VM and removing

sensitive information from the memory of the process (Section 5.2). By using these security services, cloud customers can learn from attack techniques, reveal vulnerabilities and misconfigurations as well as reveal the intentions of attackers while their original systems remain unaffected.

Finally, we proposed two security services in the field of transparent attack mitigation that operate in a pro-active fashion (Chapter 6). The first security service dynamically locates and removes sensitive data like password hashes or secret keys from a customer VM in order to store this data in a secured environment. The sensitive data is inserted "on-the-fly" into a running process in the volatile memory once it is required, only if certain security checks could be verified and only as long as it is required (Section 6.1). This way, the service greatly minimizes the probability of sensitive data being stolen by an attacker who successfully exploits software vulnerabilities on a customer system in order to bypass permission mechanisms. The second proposed security service autonomously analyzes the operating systems of customers, computes custom-made and optimized security settings for each system depending on used software and on the scenario in which the system runs. New settings are transparently inserted into the running processes with the help of different techniques and without any interruptions. No customer interaction is required. Optimal settings can change over time and our service can periodically analyze a customer setup and adapt the security settings. We called this strategy "hot-hardening". Furthermore, we introduced Dynamic Child Adaptation (DCA) which enables the proposed service to rapidly insert specific settings for new child processes depending on the task they handle.

All our proposed security services have been implemented and evaluated under realistic circumstances and the results showed the feasibility of our techniques and strategies. With the help of our novel services, the security of the customer systems in an IaaS cloud scenario can be greatly increased.

In the near future, cloud computing resources will increasingly be used by lightweight hardware devices for outsourcing and analyzing large amounts of data as well as for deploying complex processes. The continuous progress in the field of software virtualization allows to provide fine-grained specific computing resources on demand. They are dynamically and rapidly provided once they are required and for low costs. Recent progress in the field of memory and storage forensic techniques and the combination of these techniques with software virtualization can be beneficially used to transparently analyze co-resident system deployed in the cloud. In this work, we intensively used these techniques to locate, interpret and manipulate raw data extracted from running virtualized systems.

However, there are a lot of open problems which need to be investigated and solved. For example, reliable automated approaches are required that bridge semantic gaps which occur when interpreting unknown extracted raw data. This is necessary to provide flexible cloud security architectures which support different software setups without the need to have precise previous information about the customer system's internals. In this work, we showed several strategies how extracted raw data can be located, analyzed, interpreted or even manipulated in order to increase the security of customer systems. However, more sophisticated and generic strategies are required which can make use out of the extracted information during run-time, especially regarding to security purposes in multi-tenancy virtualized environments. Furthermore, there is a lot of potential in adapting existing security architectures and approaches to the "as-a-Service" business model. Cloud customers do not want to install or manage security software or settings, often they do not have the required expertise as well. Flexible and scalable security services, which generically support a wide range of different software setups and which can be simply bought by the customers in addition to other cloud services have great potential for research as well as for economy.

BIBLIOGRAPHY

[1] S. Abt and H. Baier. Towards efficient and privacy-preserving network-based botnet detection using netflow data. In *International Network Conference (INC)*, 2012. doi: 10.1109/HPCC.2009.97.

[2] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey. Technical report, University of Cambridge, Computer Laboratory, 2005. URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-641.pdf.

[3] R. Ando, Z. Zong-Hua, Y. Kadobayashi, and Y. Shinoda. A dynamic protection system of web server in virtual cluster using live migration. In *Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 95–102, 2009. doi: 10.1109/DASC.2009.154.

[4] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker-misdirection, deception, and disinformation. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 942–953, 2014. URL http://doi.acm.org/10.1145/2660267.2660329.

[5] H. Artail, H. Safa, M. Sraj, I. Kuwatly, and Z. Al-Masri. A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks. *Computer Security*, 25:274–288, 2006. URL http://dx.doi.org/10.1016/j.cose.2006.02.009.

[6] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, 2011. doi: 10.1145/2007183.2007189.

[7] M. Balamurugan and B. S. C. Poornima. Honeypot as a service in cloud. In *IJCA Proceedings of the International Conference on Web Services Computing (ICWSC)*, number 1, pages 39–43, 2011.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, 2003. URL http://doi.acm.org/10.1145/945445.945462.

[9] D. Bartholomew. Qemu: A multihost, multitarget emulator. *Linux Journal*, (145), 2006. URL http://dl.acm.org/citation.cfm?id=1134160.1134163.

[10] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, volume 15 of *USENIX-SS'06*, 2006. URL http://dl.acm.org/citation.cfm?id=1267336.1267357.

[11] S. Biedermann and J. Szefer. Systemwall: An isolated firewall using hardware-based memory introspection. In *Proceedings of the 17th Information Security Conference (ISC)*, ISC'14, 2014.

[12] S. Biedermann, M. Zittel, and S. Katzenbeisser. Improving security of virtual machines during live migrations. In *Proceeding of the 11h Annual International Conference on Privacy, Security and Trust (PST)*, pages 352–357, 2013. doi: 10.1109/PST.2013.6596088.

[13] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the zeus botnet crimeware toolkit. In *Proceedings of the 8th Annual International Conference on Privacy Security and Trust (PST)*, pages 31–38, 2010. doi: 10.1109/PST.2010.5593240.

[14] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE'07, pages 169–179, 2007. URL http://doi.acm.org/10.1145/1254810.1254834.

[15] R. Budiarto, A. Samsudin, C. W. Heong, and S. Noori. Honeypots: why we need a dynamics honeypots? In *Proceedings of the International Conference on Information and Communication Technologies: From Theory to Applications*, pages 565–566, 2004. doi: 10.1109/ICTTA.2004.1307887.

[16] L. Chen, Z. Li, C. Gao, and L. Liu. Dynamic forensics based on intrusion tolerance. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 469–473, 2009. doi: 10.1109/ISPA.2009.66.

[17] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design &*

Implementation, volume 2 of *NSDI'05*, pages 273–286, 2005. URL http://dl.acm.org/citation.cfm?id=1251203.1251223.

[19] M. Crawford and G. Peterson. Insider threat detection using virtual machine introspection. In *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, pages 1821–1830, 2013. doi: 10.1109/HICSS.2013.278.

[20] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'06, pages 581–590, 2006. doi: 10.1145/1124772.1124861.

[21] B. Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digit. Investig.*, 4:62–64, 2007. doi: 10.1016/j.diin.2007.06.008.

[22] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *IEEE Symposium on Security and Privacy (SP)*, pages 297–312, 2011. doi: 10.1109/SP.2011.11.

[23] U. Erlingsson. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV*, pages 92–134. 2007. URL http://dl.acm.org/citation.cfm?id=1793914.1793919.

[24] N. Falliere, L. O. Murchu, and E. Chien. W32.stuxnet dossier. In *Symantec Security Response*, 2010.

[25] A. Forget, S. Chiasson, P. C. van Oorschot, and R. Biddle. Improving text passwords through persuasion. In *Proceedings of the 4th symposium on Usable privacy and security*, SOUPS'08, pages 1–12, 2008. doi: 10.1145/1408664.1408666.

[26] N. Fotiou, G. F. Marias, and G. C. Polyzos. Fighting phishing the information-centric way. In *5th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2012. doi: 10.1109/NTMS.2012.6208747.

[27] T. Fraser, M. R. Evenson, and W. A. Arbaugh. Vici virtual machine introspection for cognitive immunity. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 87–96, 2008. doi: 10.1109/ACSAC.2008.33.

[28] A. Y. Fu, W. Liu, and D. Xiaotie. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *IEEE Transactions on Dependable and Secure Computing*, 3(4):301–311, 2006. doi: 10.1109/TDSC.2006.50.

[29] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium*, pages 191–206, 2003.

[30] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 193–206, 2003. URL http://doi.acm.org/10.1145/945445.945464.

[31] M. Gondree and Z. N. J. Peterson. Geolocation of data in the cloud. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY'13, pages 25–36, 2013. doi: 10.1145/2435349.2435353.

[32] The Conficker Working Group. Lessons learned. 2011.

[33] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems*, SRDS'11, pages 147–156, 2011. doi: 10.1109/SRDS.2011.26.

[34] C. Hecker, K. L. Nance, and B. Hay. Dynamic honeypot construction. In *Proceedings of the 10th Colloquium for Information Systems Security Education*, pages 95–102, 2006.

[35] J. Hong. The state of phishing attacks. *Commun. ACM*, 55(1): 74–81, 2012. doi: 10.1145/2063176.2063197.

[36] H. Huang, W. Tsai, and Y. Chen. Autonomous hot patching for web-based applications. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 51–56, 2005.

[37] Q. Huang, F. Gao, R. Wang, and Z. Qi. Power consumption of virtual machine live migration in clouds. In *Proceedings of the 3rd International Conference on Communications and Mobile Computing*, CMC'11, pages 122–125, 2011. URL http://dx.doi.org/10.1109/CMC.2011.62.

[38] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. Cloudsec: A security monitoring appliance for virtual machines in the iaas cloud model. In *Proceedings of the 5th International Conference on Network and System Security (NSS)*, pages 113–120, 2011. doi: 10.1109/ICNSS.2011.6059967.

[39] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, SSYM'04, pages 13–13, 2004. URL http://dl.acm.org/citation.cfm?id=1251375.1251388.

[40] M. Kandias, N. Virvilis, and D. Gritzalis. The insider threat in cloud computing. In *Critical Information Infrastructure Security*, volume 6983 of *Lecture Notes in Computer Science*, pages 93–103. 2013. ISBN 978-3-642-41475-6. doi: 10.1007/978-3-642-41476-3_8.

[41] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '09, pages 524–533, 2009. doi: 10.1109/DSN.2009.5270298.

[42] M. Khonji, Y. Iraqi, and A. Jones. Phishing detection: A literature survey. *IEEE Communications Surveys Tutorials*, 15(4):2091–2121, 2013. doi: 10.1109/SURV.2013.032213.00009.

[43] P. Knickerbocker, D. Yu, and J. Li. Humboldt: A distributed phishing disruption system. In *eCrime Researchers Summit*, eCRIME'09, pages 1–12, 2009. doi: 10.1109/ECRIME.2009.5342620.

[44] S. Kulkarni, M. Mutalik, P. Kulkarni, and T. Gupta. Honeydoop - a system for on-demand virtual high interaction honeypots. In *International Conferece For Internet Technology And Secured Transactions*, pages 743–747, 2012.

[45] Y. Kuno, K. Nii, and S. Yamaguchi. A study on performance of processes in migrating virtual machines. In *Proceedings of the 10th International Symposium on Autonomous Decentralized Systems*, ISADS'11, pages 567–572, 2011. URL http://dx.doi.org/10.1109/ISADS.2011.79.

[46] I. Kuwatly, M. Sraj, Z. Al Masri, and H. Artail. A dynamic honeypot design for intrusion detection. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS)*, pages 95–104, 2004. doi: 10.1109/PERSER.2004.1356776.

[47] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys'09, pages 1–12, 2009. doi: 10.1145/1519065.1519067.

[48] F. Leder and T. Werner. Know your enemy: Containing conficker. The Honeynet Project, 2009.

[49] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias. Virtual machine introspection in a hybrid honeypot architecture. In *Proceedings of the 5th USENIX conference on Cyber Security Experimentation and Test*, CSET'12, pages 5–5, 2012. URL http://dl.acm.org/citation.cfm?id=2372336.2372343.

[50] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 164–177. 2013. ISBN 978-3-642-38630-5. doi: 10.1007/978-3-642-38631-2_13.

[51] T. Li, F. Han, D. Shuai, and Z. Chen. Larx: Large-scale anti-phishing by retrospective data-exploring based on a cloud computing platform. In *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–5, 2011. doi: 10.1109/ICCCN.2011.6005822.

[52] X. Li, C. Zhang, X. Lin, and S. Lin. Vad: A detail investigation into process's memory. In *Proceedings of the International Conference on Computational Intelligence and Security*, volume 1 of *CIS'09*, pages 531–536, 2009. URL http://dx.doi.org/10.1109/CIS.2009.130.

[53] Z. Liu, W. Qu, W. Liu, and K. Li. Xen live migration with slowdown scheduling algorithm. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'10, pages 215–221, 2010. URL http://dx.doi.org/10.1109/PDCAT.2010.88.

[54] R. Mallikka, A. A. Saleh, and S. Putra. Prevention of phishing attacks based on discriminative key point features of webpages. *International Journal of Computer Science and Security (IJCSS)*, pages 1–18, 2012.

[55] J. N. Matthews, E. M. Dow, T. Deshane, W. Hu, J. Bongio, P. F. Wilbur, and B. Johnson. *Running Xen: A Hands-On Guide to the Art of Virtualization*. 1 edition, 2008.

[56] E. Medvet, E. Kirda, and C. Kruegel. Visual-similarity-based phishing detection. In *Proceedings of the 4th international conference on Security and privacy in communication networks*, SecureComm'08, pages 22:1–22:6, 2008. doi: 10.1145/1460877.1460905.

[57] A. More and S. Tapaswi. Dynamic malware detection and recording using virtual machine introspection. In *Best Practices Meet (BPM)*, pages 1–6, 2013. doi: 10.1109/BPM.2013.6615011.

[58] S. Mrdovic, A. Huseinovic, and E. Zajko. Combining static and live digital forensic analysis in virtual environment. In *22nd International Symposium on Information, Communication and Automation Technologies (ICAT)*, pages 1–6, 2009. doi: 10.1109/ICAT.2009.5348415.

[59] J. Oberheide, E. Cooke, and F. Jahanian. Exploiting live virtual machine migration. In *BlackHat DC Briefings*, 2008.

[60] J. Owens and J. Matthews. A study of passwords and methods used in brute-force ssh attacks, 2008.

[61] P. Pal and M. Atighetchi. The phishbouncer experience. In *Cybersecurity Applications Technology Conference For Homeland Security*, CATCH'09, pages 150–154, 2009. doi: 10.1109/CATCH.2009.12.

[62] M. Payer and T. R. Gross. Hot-patching a web server: A case study of asap code repair. In *Proceedings of the 11th Annual International Conference on Privacy, Security and Trust (PST)*, pages 143–150, 2013.

[63] B. D. Payne and W. Lee. Secure and flexible monitoring of virtual machines. In *Annual Computer Security Applications Conference*, pages 385–397, 2007. doi: 10.1109/ACSAC.2007.38.

[64] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'08, pages 233–247, 2008. doi: 10.1109/SP.2008.24.

[65] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th conference on USENIX Security Symposium*, volume 13 of *SSYM'04*, 2004. URL http://dl.acm.org/citation.cfm?id=1251375.1251376.

[66] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, 2007. ISBN 0321336321.

[67] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto. Katana: A hot patching framework for elf executables. In *Proceedings of the International Conference on Availability, Reliability, and Security (ARES)*, pages 507–512, 2010.

[68] S. Rauti and V. Leppänen. Browser extension-based man-in-the-browser attacks against ajax applications with countermeasures. In *Proceedings of the 13th International Conference on Computer Systems and Technologies*, CompSysTech'12, pages 251–258, 2012. URL http://doi.acm.org/10.1145/2383276.2383314.

[69] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *Proceedings of the International Conference on VLSI Design*, pages 605–611, 2004.

[70] C. Rong, S. T. Nguyen, and M. G. Jaatun. Beyond lightning: A survey on security challenges in cloud computing. *Computers and Electrical Engineering*, 39(1):47–54, 2013. URL http://dx.doi.org/10.1016/j.compeleceng.2012.04.015. Special issue on Recent Advanced Technologies and Theories for Grid and Cloud Computing and Bio-engineering.

[71] A. P. E. Rosiello, E. Kirda, C. Kruegel, and F. Ferrandi. A layout-similarity-based approach for detecting phishing pages. In *SecureComm*, pages 454–463, 2007.

[72] A. Sardana and R. C. Joshi. Autonomous dynamic honeypot routing mechanism for mitigating DDoS attacks in DMZ. In *16th IEEE International Conference on Networks (ICON)*, pages 1–7, 2008. doi: 10.1109/ICON.2008.4772623.

[73] A. Satoh, Y. Nakamura, and T. Ikenaga. Ssh dictionary attack detection based on flow analysis. In *IEEE/IPSJ 12th International Symposium on Applications and the Internet (SAINT)*, pages 51–59, 2012. doi: 10.1109/SAINT.2012.16.

[74] S. Sheng, M. Holbrook, P. Kumaraguru, L. F. Cranor, and J. Downs. Who falls for phish?: a demographic analysis of phishing susceptibility and effectiveness of interventions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'10, pages 373–382, 2010. doi: 10.1145/1753326.1753383.

[75] sKyWIper Analysis Team. skywiper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks, 2012.

[76] A. K. Sood, R. J. Enbody, and R. Bansal. Dissecting spyeye: Understanding the design of third generation botnets. *Computer Networks*, 57(2):436–450, 2013. doi: 10.1016/j.comnet.2012.06.021.

[77] S. G. Soriga and M. Barbulescu. A comparison of the performance and scalability of xen and kvm hypervisors. In *Proceedings of the 12th International Conference on Networking in Education and Research (RoEduNet)*, pages 1–6, 2013. doi: 10.1109/RoEduNet.2013.6714189.

[78] J. M. Stanton, K. R. Stam, P. Mastrangelo, and J. Jolton. Analysis of end user security behaviors. *Computers and Security*, 24(2):124–133, 2005. URL http://dx.doi.org/10.1016/j.cose.2004.07.001.

[79] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS'03, pages 160–171, 2003. URL http://doi.acm.org/10.1145/782814.782838.

[80] B. Sun, Q. Wen, and X. Liang. A dns based anti-phishing approach. In *Proceedings of the 2nd International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC)*, volume 2, pages 262–265, 2010. doi: 10.1109/NSWCTC.2010.196.

[81] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast live cloning of virtual machine based on xen. In *11th IEEE International Conference on High Performance Computing and Communications*, HPCC'09, pages 392–399, 2009. doi: 10.1109/HPCC.2009.97.

[82] J. L. Thames, R. Abler, and D. Keeling. A distributed active response architecture for preventing ssh dictionary attacks. In *IEEE Southeastcon*, pages 84–89, 2008. doi: 10.1109/SECON.2008.4494264.

[83] M. Tkalcic and J. F. Tasic. Colour spaces: perceptual, historical and applicational background. In *EUROCON 2003. Computer as a Tool.*, pages 304–308, 2003. doi: 10.1109/EURCON.2003.1248032.

[84] N. Utakrit. A review of browser extensions, a man-in-the-browser phishing techniques targeting bank customers. In *Proceedings of the Australian Information Security Management Conference*, 2009.

[85] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM symposium on Operating systems principles*, SOSP'05, pages 148–162, 2005. doi: 10.1145/1095810.1095825.

[86] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. *SIGCOMM Comput. Commun. Rev.*, 38 (4):231–242, 2008. URL http://doi.acm.org/10.1145/1402946.1402985.

[87] B. Wei, C. Lin, and X. Kong. Energy optimized modeling for live migration in virtual data center. In *Proceedings of teh International Conference on Computer Science and Network Technology (ICCSNT)*, volume 4, pages 2311–2315, 2011. doi: 10.1109/ICCSNT.2011.6182436.

[88] J. S. White, J. N. Matthews, and J. L. Stacy. A method for the automated detection phishing websites through both site characteristics and image analysis. In *Society of Photo-Optical Instrumentation Engineers (SPIE)*, volume 8408, 2012. doi: 10.1117/12.918956.

[89] T. Yashiro, M. Bessho, S. Kobayashi, N. Koshizuka, and K. Sakamura. T-kernel/ss: A secure filesystem with access control protection using tamper-resistant chip. In *IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, pages 134–139, 2010.

[90] W. D. Yu, S. Nargundkar, and N. Tiruthani. A phishing vulnerability analysis of web based systems. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 326–331, 2008. doi: 10.1109/ISCC.2008.4625681.

[91] C. Yue and H. Wang. Anti-phishing in offense and defense. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 345–354, 2008. doi: 10.1109/ACSAC.2008.32.

[92] S. Zawoad, A. K. Dutta, and R. Hasan. Seclaas: Secure logging-as-a-service for cloud forensics. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS'13, pages 219–230, 2013. URL http://doi.acm.org/10.1145/2484313.2484342.

[93] F. Zhang, Y. Huang, H. Wang, H. Chen, and B. Zang. Palm: Security preserving vm live migration for systems with vmm-enforced protection. In *Proceedings of the 3rd Asia-Pacific Trusted Infrastructure Technologies Conference*, APTC '08, pages 9–18, 2008. doi: 10.1109/APTC.2008.15.

[94] X. Zhang and Y. Dong. Optimizing xen vmm based on intel(r) virtualization technology. In *Proceedings of the International Conference on Internet Computing in Science and Engineering*, ICICSE'08, pages 367–374, 2008. doi: 10.1109/ICICSE.2008.81.

[95] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER'10, pages 88–96, 2010. URL http://dx.doi.org/10.1109/CLUSTER.2010.17.

[96] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS'12, pages 305–316, 2012. URL http://doi.acm.org/10.1145/2382196.2382230.

[97] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: experiment-based management of virtualized data centers. In *Proceedings of the USENIX Annual technical conference*, USENIX'09, pages 18–18, 2009. URL http://dl.acm.org/citation.cfm?id=1855807.1855825.

WISSENSCHAFTLICHER WERDEGANG

Oktober 2004 - November 2010

Studium der Informatik an der Universität Würzburg

Dezember 2010 - Februar 2015

Promotion an der TU Darmstadt unter Leitung von Prof. Dr. Stefan Katzenbeisser

Dezember 2010 - Mai 2015

Wissenschaftlicher Mitarbeiter am Fachgebiet "Security Engineering Group" der Technischen Universität Darmstadt